Journal of Global Economic Analysis, Volume 11 (2026), No. 01, pp. 86-137.

MPSGE in Julia: Bringing Open-Source and Efficient Computation to concise CGE Model formation

BY ELI LAZARUS[a], MITCH PHILLIPSON[b], DAVID ANTHOFF[c], AND THOMAS RUTHERFORD[d]

This paper introduces MPSGE.jl, a free and open-source package in Julia that facilitates CGE model building from tabular definitions. The package's programmatically generated equations reduce redundant and repetitive code, errors, and development time. The built-in functions simplify troubleshooting, analysis, and reports. Embedding within a general-use scientific language enables streamlined integration with general-use functions and tools. The design combines computational efficiency with intuitive syntax and model formation. The open-source foundation allows for greater access, and user contributions to ongoing development. This work is a contribution to the open-science movement, with the aim of increasing access, robustness, transparency, and collaboration. We give an overview of how to use the package, its construction, and highlight some of its features. We include example models of different forms and scale as illustrations. First, we employ a simple toy model to introduce the basic structure. Then, we illustrate how the flexibility of MPSGE in Julia can facilitate functionality not possible in the GAMS progenitor. With a third example model, we demonstrate the package used at scale by evaluating tariff effects using a model with five household types for each U.S. state. We link to the package and its documentation for further reading and utilization.

JEL codes: D58, C63, C68, C88, L17

Keywords: CGE, MPSGE, MPSGE.jl, Open-Source

---

[a] University of California, Berkeley, Giannini Hall, Berkeley, CA 94720 (e-mail: Lazarus.eli@berkeley.edu).
[b] University of Wisconsin - Madison, 204 Taylor Hall, 427 Lorch St., Madison, WI 53706 (e-mail: mphillipson@wisc.edu).
[c] University of California, Berkeley, 345 Giannini Hall, Berkeley, CA 94720 (e-mail: anthoff@berkeley.edu)
[d] University of Wisconsin – Madison, Taylor Hall, 427 Lorch St., Madison, WI 53706 (e-mail: rutherford@aae.wisc.edu)

1. Introduction

In this paper, we introduce and describe MPSGE.jl, a novel, open-source platform for succinct-form Computable General Equilibrium (CGE) model building. MPSGE.jl is a package developed in the open-source scientific programming language, Julia. It facilitates simple, fast, accessible, adaptable, and less error-prone CGE model development. The package generates the appropriate sets of equations programmatically from user-defined, tabular[1] representations of relevant economic elements and data.

MPSGE.jl is an evolution of the Mathematical Programming System for General Equilibrium (MPSGE/GAMS) (Rutherford 1987; 1999). It provides a structure to effectively and efficiently construct CGE models following Walras' Law (Lange 1942; Mathiesen 1985; Scarf and Hansen 1973), with enough flexibility to represent a wide swath of general equilibrium forms.

MPSGE/GAMS implements Mathiesen's format for economic equilibrium modelling (Mathiesen 1985), which structures the general equilibrium system as a set of connected variables and complementary functions. MPSGE/GAMS extends Orchard Hayes' MPS format for Linear Programming which generates systems of equations from tabular input files, applied to Arrow-Debreu economic general equilibria. It was originally written in Fortran, and Rutherford later established the program as an embedded subsystem of the domain-specific[2] GAMS (Meeraus 1983) programming language.

The MPSGE.jl package offers a number of advantages relative to other common tools for CGE modelling, bringing the advantages of Rutherford's MPSGE/GAMS into a free, open-source, general-use scientific language, with additional functionality, accessibility, flexibility, computational efficiency, and increased potential for ongoing development and collaborative workflows[3]. The intention here is to give an overview of the platform, and a selection of key features. Comprehensive documentation is beyond the scope of this paper.

In the following sections, we give more detail on the package and its functionality and further demonstrate its usefulness with example models. First, we will give an overview of how a model is constructed with the package. Then we provide a basic maquette CGE model in Julia[4] to introduce readers to the

---

[1] The display and syntax are not explicitly in table form, but each entry consists of values corresponding to a fixed set of fields.

[2] GAMS (general algebraic modeling system) is formulated for mathematical optimization.

[3] The PATH solver which is embedded in MPSGE.jl requires a license for large models. H owever, it is free for non-commercial use. In fact, no specific restriction beyond temporar y nature of the free license is articulated on the website (https://pages.cs.wisc.edu/~ferri s/path.html), but users are advised to email for information on obtaining an unrestricted license.

[4] Making use of the general optimization package in Julia, JuMP.jl

language, and in parallel show the same model in MPSGE.jl and MPSGE/GAMS form. Following, we discuss the syntax and functionality of the package, highlighting just some of the features. In the subsequent section, we demonstrate one example of the MPSGE.jl version's flexibility relative to its GAMS-based predecessor with an example model looking at internal migration responses to regional economic changes. We then show an example of a large dimensional dataset model, simulating the distributional effects of tariffs on multiple household types for each U.S. state. We also use the U.S. state model to benchmark test the computational speed of the package. A discussion section focuses on ongoing development. The final section concludes.

2. Structure of an MPSGE.jl model

The core of generating a CGE model using MPSGE.jl consists of constructing a set of production[5] and final demand blocks, defining the flow of commodities from factor inputs through to final consumption. A production block associates a set of commodities as inputs to a sector's[6] production, linked to a set of output commodities produced from those inputs by that sector. A final demand block associates a set of commodities as initial endowments[7] provided by a consumer agent, along with a set of commodities demanded for final consumption. Each input, output, initial endowment and final demand commodity has a total value specified in currency terms, coming from the appropriate economic data. The set of production and demand blocks constitute the core general equilibrium structure, which can be augmented with additional detail and extended with auxiliary elements. Table 1 summarizes the four main steps for constructing a model in MPSGE.jl.

For each production activity, the program automatically generates cost and revenue equations, which are then used to construct the zero profit condition constraints. Cost equations are generated for every set of inputs with a Constant Elasticity of Substitution (CES) functional form[8], using elasticity parameter values supplied by the modeler. Revenue equations for every set of outputs are generated in Constant Elasticity of Transformation (CET) form.

Inputs can be nested by user definition, such that they are combined into composite inputs at the level above. The program will automatically construct

---

[5] A production block defines a set of cost and revenue functions associated with the production activity of a particular sector. It is not a production function.

[6] It is common to think of a sector as a representative firm.

[7] Commonly including primary factors such as labor and capital.

[8] An elasticity parameter value of 0 collapses in the CES equation to generate Leontief complements. An elasticity value of 1 automatically generates the Cobb-Douglas functional form.

nested CES functions, with specific elasticity values for each nest level. Output distributions can be nested in a parallel manner.

Taxes can be set on any input or output by providing a rate and designating a consumer to receive the revenue. The equations written by the package will automatically distribute the revenue appropriately.

Auxiliary variables and constraints of flexible form can also be included, which provides scope for a multiplicity of structures and extensions.

Table 1. Four central components for constructing a model in MPSGE.jl.

Component	Description
1	Declare parameters, sectors, commodities, consumers, and auxiliary variables
2	For each sector create a Production block which defines the flow of commodities into and out of the sector
3	For each consumer create a Demand block which defines the consumer's endowments and final demands
4	Create additional constraints for auxiliary variables

Source: Authors

Once the model is defined with all variables and blocks, the primary CGE equilibrium conditions of zero profit for each sector and market clearance for each commodity, along with the income balance identity for each consumer, are automatically constructed by the program in the build step. Zero profit constraints set the total value of the inputs gross of taxes equal to the total value of outputs net of taxes for every sector. Market clearance constraints set the total value of supply of each commodity equal to the total value of its demand. The income balance constraint sets the total value of demands equal to the total value of endowments and tax revenues for every consumer.

2.1 Introductory Model: CGE in Julia (JuMP)

To introduce the fundamentals of using the MPSGE.jl package, we begin with a small example model[9] rendered in Julia[10] without the package, before illustrating the efficiencies and syntax when generating the same model with MPSGE.jl. We include an MPSGE/GAMS version for comparison. Appendix A includes a fuller description of this toy model with full copies of all three versions and the results

[9] This toy model builds from one of Markusen's many mpsge examples, which are available at https://www.mpsge.org/markusen/m2.htm#m2_2s.

[10] A detailed introduction to Julia is beyond the scope of this paper. The language is well-documented with an active and responsive user and developer community. The syntax is designed to be intuitive for scientific computation, and should look familiar to those who know R or Python, or even Matlab.

of a sample counterfactual simulation. The GitHub repository [11] and documentation[12] have more examples, including a number built from GAMS-based models and comparing MPSGE/GAMS to MPSGE.jl[13].

To generate the model in Julia, we will make use of JuMP.jl[14], the Julia for Mathematical Programming package. JuMP is an algebraic modelling language embedded within Julia, giving a mathematical interface for optimization problems (Dunning et al. 2017; Lubin et al. 2023). JuMP.jl functions similarly to the GAMS language, and is useful for a range of optimization types. The MPSGE.jl package uses JuMP.jl as a core dependency, while providing special functionality and a pre-defined but still flexible structure suited to CGE.

2.2 Declaring the Model and Variables: Julia JuMP, MPSGE.jl, and MPSGE/GAMS

We start with a set up step. For the Julia JuMP version, assuming we have Julia installed and have downloaded the two packages we need - JuMP.jl and PATHSolver.jl - into our Julia packages library, we first activate those packages with the 'using' statement. To construct an optimization model with JuMP, we first declare a model, giving it a name, and optionally select our preferred solver at the same time[15]. Next, we declare our variables[16] in a group with @variables, then model name, and a begin/end block that lists all the variables. We declare two tax rates as parameters[17], set initially at 0%.

```
### Julia JuMP mcp
using JuMP, PATHSolver
mcp = Model(PATHSolver.Optimizer)
    @variables(mcp, begin
        X, (start = 1)
        Y, (start = 1)
```

[11] https://github.com/julia-mpsge/MPSGE.jl/tree/main/examples is the section of the MPSGE.jl GitHub repository with example models.
[12] https://julia-mpsge.github.io/MPSGE.jl/stable/ is the documentation main page for the MPSGE.jl package, integrated with the GitHub repository.
[13] https://github.com/julia-mpsge/MPSGE.jl/tree/main/examples/Compare_Julia_to_GAMS is a subsection of the MPSGE.jl repository that has some examples comparing MPSGE.jl versions to GAMS versions of some example models.
[14] https://jump.dev/JuMP.jl is the documentation main page for the JuMP.jl package, integrated with the GitHub repository.
[15] PATHSolver is a package that wraps the PATH solver for ease of use with Julia. https://github.com/chkwon/PATHSolver.jl, and https://pages.cs.wisc.edu/~ferris/path.html is the GitHub repository for the PATHSolver.jl package.
[16] The variable names here are chosen by MPSGE/GAMS convention, using single uppercase letters for sector names, and P (price) with each commodity. Julia is case sensitive.
[17] A JuMP parameter stores a value with a name as part of the model so that the value can be updated later for counterfactual simulations without having to re-define the entire model.

```
        W, (start = 1)
        PX, (start = 1)
        PY, (start = 1)
        PW, (start = 1)
        PL, (start = 1)
        PK, (start = 1)
        CONS, (start = 200)
        tax_pl in JuMP.Parameter(0)
        tax_pk in JuMP.Parameter(0)
    end)
```

In the two columns below, we show the same set up step for the MPSGE.jl and MPSGE/GAMS versions of the model. For the MPSGE.jl version, first we activate the package[18] with the using statement, then we initiate and name our model. We define model parameters within the model, as with JuMP.

In the GAMS version, parameters are set outside the model, then the model and MPSGE elements are defined within an $ONTEXT block.

Declaring our model variables with MPSGE.jl and MPSGE/GAMS[19] is similar to JuMP. Here, we declare them within the pre-structured Mathiesen format as sectors, commodities, and consumers. Sector activity levels and commodity prices have a default start value of 1 so in this model we do not need to specify those. The benchmark consumer start value will be included in the final demand block.

```
### MPSGE.jl version
using MPSGE
M = MPSGEModel()
@parameters(M, begin
    tax_pl, 0
    tax_pk, 0
end)

@sectors(M, begin
    X
    Y
    W
end)
@commodities(M, begin
    PX
    PY
    PW
    PK
    PL
end)
@consumer(M, CONS)
```

```
* GAMS version
PARAMETERS
    TX_PL,
    TX_PK;
TX_PL = 0;
TX_PK = 0;

$ONTEXT
$MODEL: M22

$SECTORS:
    X
    Y
    W
$COMMODITIES:
    PX
    PY
    PL
    PK
    PW
$CONSUMERS:
    CONS
```

Looking at the MPSGE.jl and MPSGE/GAMS versions in parallel shows the close similarity in syntax.

[18] We do not need to activate PATHSolver or JuMP with a using statement here even though they are used because they are linked implicitly as dependencies of the MPSGE.jl package.
[19] GAMS is not case sensitive. Upper case is used here for variable names by convention.

2.3 Constructing the Constraints and Equilibrium Conditions

2.3.1 Constraints: Sector X Production Block for each Version

Here we first show the construction of a single production block for each version.

In this model, an X sector combines 40 units of input commodity labor (PL), 60 of capital (PK), and 20 of commodity PY to produce 120 units of the output commodity PX. The labor and capital inputs are combined in a nest with a Cobb-Douglas functional form into a composite value-added commodity (va). Input taxes are set on both labor and capital, initially with a value of 0%. The composite commodity 'va' is combined with PY in the top level CES nest "s". The output nest "t" is set as a Leontief functional form[20]. Revenues from both taxes go to the consumer 'CONS'. All benchmark reference prices are set to $1 per unit[21] by default.

In this Julia JuMP mcp[22] version, we separate out the elements in the way that they are constructed by MPSGE.jl 'under the hood' to illustrate the underlying structure. At the bottom level of the inputs, the outer leaf node[23] unit cost functions for labor and capital[24] incorporate the tax parameters, circled here to highlight all the places they appear in the full algebraic mcp equations compared to the other versions. The unit cost function for the va nest then combines labor and capital in Cobb-Douglas form, with the share of each input into that nest as the exponents.

```
### Julia JuMP mcp
cost_X_va_PL = PL*(1+tax_pl))
cost_X_va_PK = PK*(1+tax_pk))
cost_X_va = cost_X_va_PL^(40/100) * cost_X_va_PK^(60/100)
```

The cost function for the PY outer leaf node input is simply PY. The top level unit cost function combines va and PY with a CES functional form and an elasticity value of 0.5 set by the user. The unit revenue function for the single output commodity here is simply PX.

```
cost_X_PY = PY
cost_X = (20/120 * cost_X_PY ^(1-.5) + 100/120 * cost_X_va^(1-.5))^(1/(1-.5))
```

---

[20] The functional forms for outputs define the distribution or transformation into multiple outputs. With only a single output, the form actually makes no difference, so here the elasticity value and the form is nominal.

[21] Or $1 billion, €1 million etc. depending on the currency unit of the input data.

[22] The name mcp here refers to the mixed complementarity problem structure. Note the perpendicular symbols defining the relations between the variables and expressions in the JuMP constraints. MPSGE.jl models are also constructed as MCPs by default.

[23] A diagram of the nesting structure resembles an upside-down branching tree. Each end of a 'branch' is a node.

[24] For this toy model, we can set these subexpressions as global variables, but that would be computationally expensive in the solution step for a large scale model.

```
revenue_X = PX
```

The total profit equation subtracts the total cost (benchmark total input quantity times the top level unit cost function) from the total revenue (benchmark total output quantity times the unit revenue function). Finally, we enter the zero profit constraint with the model name, the constraint name, and the equation set as complementary[25] to the sector activity level variable. The equation is set to solve to zero by default.

```
profit_X = 120*revenue_X - 120*cost_X
@constraints(mcp, zp_X, -profit_X ⊥ X)
```

For the sector X production block in the MPSGE.jl version, we simply define the input, output, and nest relationships, the elasticity values, and the benchmark quantities. The cost and revenue functions and the zero profit equation will be automatically constructed by the program. Here, we set the output elasticity as 0[26], the top input nest as CES with an elasticity of 0.5, and the va nest (which feeds into the top "s" input nest) as Cobb-Douglas by setting the elasticity as 1. We apply the taxes to the respective input factor commodities, and specify the consumer to receive any revenues. Note that the taxes are simply linked once to the PL and PK inputs for sector 'X'. Applying the tax within that production block is sufficient for that term to appear appropriately throughout the set of constructed constraint equations in the model, as shown in the mcp model.

```
### MPSGE.jl
@production(M, X, [t=0, s=.5, va=>s=1], begin
    @output(PX, 120, t)
    @input(PY, 20, s)
    @input(PL, 40, va, taxes=[Tax(CONS,tax_pl)])
    @input(PK, 60, va, taxes=[Tax(CONS,tax_pk)])
end)
```

The MPSGE/GAMS version is parallel to the MPSGE.jl above. The output nest elasticity[26] is set to 0 (Leontief) by default. The "s" nest name is reserved for the top level nest, set to 1 (Cobb-Douglas) by default so here we set it articulately as 0.5. In MPSGE.jl, there are no hidden default elasticity values and all must be articulately assigned, and the nest names are all optional, so long as the nest relationships are well defined.

```
* GAMS
$PROD:X s:0.5 va:1
    O:PX Q:120
    I:PY Q: 20
    I:PL Q: 40 va: A:CONS T:TX_PL
```

[25] Each constraint equation linked to each variable will have a solved (marginal) value. In the complementary structure, either the marginal condition or the variable must be approximately zero; therefore, a zero variable does not need to have a zero marginal value.
[26] An elasticity value set to 0 constructs a Leontief/complements functional form for that nest. Here, the value and functional form is nominal because there is only a single output.

```
    I:PK Q: 60 va: A:CONS T:TX_PK
```

2.3.2 Constraints: Remainder of the Model Construction for each Version

Below, we show the constraints that build the rest of the model for each version. We have a second sector, Y, with labor and capital nested in Cobb-Douglas form, then combined in a CES form with PX to generate output commodity PX. A W sector tracks Hicksian welfare, combining PX and PY in Cobb-Douglas form to output PW that is then consumed as final demand in the demand block in exchange for labor and capital supplied by the consumer CONS.

For the full, algebraic version immediately below, we construct the zero profit equations for each sector, the market clearance equations for each commodity, and the income balance equations for each consumer. Variables, cost functions, and parameters appear multiple times through the set of equations.

```
### Julia JuMP mcp
cost_Y_va = (PL)^(60/100) * (PK)^(40/100)
cost_Y = (20/120 * PX^(1-.75) + 100/120 * cost_Y_va^(1-.75))^(1/(1-.75))
revenue_Y = PY

cost_W = PY^.5*PX^.5
revenue_W = PW

# Zero Profit
@constraints(mcp, begin
  zp_Y, 120*cost_Y - 120*revenue_Y ⊥ Y
  zp_W, 200*cost_W - 200*revenue_W ⊥ W
  end)

# Market Clearance
@constraints(mcp, begin
  mc_PX, 120*X - 20*Y*(cost_Y/PX)^.75 - 100*W*(cost_W/PX) ⊥ PX
  mc_PY, -20*X*(cost_X/PY)^.5 + 120*Y - 100*W*(cost_W/PY) ⊥ PY
  mc_PW, 200*W - CONS/W ⊥ PW
  mc_PL, -40*X*(cost_X/cost_X_va)^.5*(cost_X_va/(PL*(1+tax_pl))) -
60*Y*(cost_Y/cost_Y_va)^.75*(cost_Y_va/PL) + 100 ⊥ PL
  mc_PK, -60*X*(cost_X/cost_X_va)^.5*(cost_X_va/(PK*(1+tax_pk))) -
40*Y*(cost_Y/cost_Y_va)^.75*(cost_Y_va/PK) + 100 ⊥ PK
    end)

# Income Balance
@constraint(mcp, ib, CONS - 100*PL - 100*PK -
40*tax_pl*PL*X*(cost_X/cost_X_va)^.5*(cost_X_va/(PL*(1+tax_pl)))  -
60*tax_pk*PK*X*(cost_X/cost_X_va)^.5*(cost_X_va/(PK*(1+tax_pk))) ⊥ CONS)
```

In the MPSGE.jl and MPSGE/GAMS versions below, we define the relationships, elasticity values, and benchmark quantities in blocks. The cost and revenue functions, all zero profit equations, market clearance equations, and income balance equations are automatically constructed by the programs. The functional forms are automatically constructed with elasticity values of "1" (Cobb-Douglas), "0" (Leontief), or other values (CES).

```
# MPSGE.jl
@production(M, Y, [t=0, s=.75, va=>s=1],
begin
```

```
* GAMS
$PROD:Y s:0.75 va:1
    O:PY Q:120
```

```
    @output(PY, 120, t)
    @input(PX, 20, s)
    @input(PL, 60, va)
    @input(PK, 40, va)
end)

@production(M, W, [t=0, s=1], begin
    @output(PW, 200, t)
    @input(PX, 100, s)
    @input(PY, 100, s)
end)

@demand(M, CONS, begin
    @final_demand(PW, 200)
    @endowment(PL, 100)
    @endowment(PK, 100)
 end)
```

```
    I:PX Q: 20
    I:PL Q: 60 va:
    I:PK Q: 40 va:

$PROD:W s:1
    O:PW Q:200
    I:PX Q:100
    I:PY Q:100


$DEMAND:CONS
    D:PW Q:200
    E:PL Q:100
    E:PK Q:100

$OFFTEXT
$SYSINCLUDE mpsgeset M22
```

Note that in MPSGE.jl and Julia JuMP, we specify the model name for each block, which enables the parallel construction of elements for separate models.

2.3.3 Reporting

Next, we show how to print some model objects in each version of the model.

For the Julia JuMP model, object_dictionary('model_name') prints a dictionary with all the model constraints and their names. The model name prints the JuMP model summary. all_constraints('model_name') prints just the constraints as a vector.

For the MPSGE.jl model, 'model_name'.object_dict prints the set of declared variables. 'model_name'.jump_model prints the JuMP model as it was constructed by the MPSGE.jl program to be sent to the JuMP package. The model name prints the entire model in MPSGE form.

```
### Julia JuMP mcp
object_dictionary(mcp)

mcp
all_constraints(mcp;
include_variable_in_set_constraints = true)
```

```
# MPSGE.jl
M.object_dict

M.jump_model

M # Print MPSGE model
```

For the MPSGE/GAMS version, running a GAMS script will automatically generate a list file that includes a print of the model, variable list, and other outputs.

2.3.4 Checking Benchmark, Solving Counterfactuals

Below we show the syntax and functions to solve each version of the model at the benchmark and to simulate an illustrative counterfactual. We increase labor and capital taxes for sector X from 0% to 50%. We show the functions for our Julia JuMP version in the left column, the MPSGE.jl version in the right column, and the MPSGE/GAMS below.

First, we fix the PW commodity to its benchmark value to serve as the numeraire price. Then we solve each model with zero iterations to check that the

data is balanced appropriately at the benchmark. A balanced set of data for the model structure will have an approximately zero postsolved value[27]. The PATH output will automatically print the postsolved residual, which is the largest residual across all variable/constraint pairs and represents the largest remaining discrepancy in the model conditions.

We save all the benchmark variable results in a dictionary for the JuMP version, and in a data frame for the MPSGE.jl version.

We then print the market clearance constraint for PX, the associated market clearance equation solution, and then the variable value as an illustration of some report options[28].

To run the counterfactual, we update the tax parameter values to 0.5 and re-solve the models without the zero iteration limit. Finally, we save all the counterfactual solutions in a dictionary or data frame.

```
# JuMP model, mcp
fix(PW,1;force=true)

set_optimizer_attribute(mcp,
"major_iteration_limit", 0)
optimize!(mcp)

Benchmarkresults_mcp = Dict(
    k => value.(v) for
    (k, v) in
object_dictionary(mcp))

mc_PX
value(mc_PX)
value(PX)

set_parameter_value(tax_pl, 0.5)
set_parameter_value(tax_pk, 0.5)

set_optimizer_attribute(mcp,
"major_iteration_limit", 10000)
optimize!(mcp)
Tax50_results_mcp = Dict(
    k => value.(v) for
    (k, v) in
object_dictionary(mcp))
```

```
# MPSGE.jl model, M
fix(PW, 1)

solve!(M, cumulative_iteration_limit=0)

Benchmarkresults_M = generate_report(M)

market_clearance(PX)
value(market_clearance(PX))
value(PX)

set_value!(M[:tax_pl], 0.5)
set_value!(M[:tax_pk], 0.5)

solve!(M)

Tax50_results_M = generate_report(M)
```

In GAMS, we run the benchmark check and the counterfactual. All variables and solutions are included in the list and/or gdx file generated when the script is run. No equations are displayed or retrievable.

```
* GAMS
PW.FX = 1;

M22.iterlim = 0;
```

---

[27] Approximately zero meaning that the absolute value is below some set tolerance, e.g. 10^{-6}.

[28] Running \`value('*constraint*')\` in JuMP prints the both the equation solution and the variable value.

```
$INCLUDE M22.GEN
SOLVE M22 USING MCP;

* Counterfactual
TX_PL = .5;
TX_PK = .5;

M22.iterlim = 10000;
$INCLUDE M22.GEN
SOLVE M22 USING MCP;
```

We can see from the above simple example that JuMP in Julia provides a platform for building and solving an array of optimization problem types, including CGE models if constructed appropriately. MPSGE.jl provides functionality on top of that platform, tailored specifically for CGE modelling, facilitating a number of significant efficiencies and conveniences that become increasingly important as model size and resolution increases. MPSGE/GAMS also generates the required CGE equilibrium condition equations from the block definitions, though they cannot be displayed. To access individual variable or other values for reporting or other use with MPSGE/GAMS, a user must access or query the lst or gdx file, or generate another output file.

In the next section, we briefly highlight a selection of features of the MPSGE.jl package before turning to two other example models.

3. MPSGE.jl Features

3.1 MPSGE.jl Functions

MPSGE.jl has a number of functions linked to the specific CGE structure. We list a subset below in Table 2, with a brief description of each. We adopt the convention of using ′′ to denote a type of model element, otherwise all text and symbols are literal. More detail from the documentation on any function is available within Julia by using the in-built help system (e.g. with ?). As with many programming languages, the output of any function can be saved to a name[29] with " 'variable_name' = [function call]".

For any of the equations or variables in the model, wrapping that function or variable in a 'value()' function will return the value from assessing that equation, or the current solution value of that variable or parameter. For example, value(market_clearance('commodity_name')) or value('sector_name').

All the functions and calls listed above can be run at any point after a model has been built and solved. It is not necessary for a user to decide ahead of building or solving a model which result values they might want to investigate or report.

[29] Julia is case sensitive and accepts a large number of Unicode symbols, including emojis (see https://docs.julialang.org/en/v1/manual/unicode-input/). Variable names cannot include spaces or start with numbers.

Table 2. A selection of MPSGE.jl functions.

Function	*Description*
'model'	Prints the entire model in MPSGE tabular form.
jump_model('model')	Returns the version of the model as passed to JuMP.
production('sector'), or *production('model'['sector'])* [a]	Returns an individual production block in MPSGE tabular form.
sectors('model') *commodities('model')* *consumers('model')* *parameters('model')* *auxiliaries('model')*	Returns all variables of the given type in a vector.
generate_report('model')	Builds a dataframe of all model variables, along with their solution values and margin values[b] from the most recently solved results.
unit_cost_function('sector'; depth = -1) *unit_cost_function('sector', 'nest'; depth = -1)*	Returns the unit cost function for the given 'sector' and 'nest'. The keyword option 'depth=' controls the depth of the production tree returned. If not specified, the default depth, -1, returns the unit cost function for the entire input tree. The 'depth =' keyword is available for every function that incorporates a unit cost function, e.g. zero_profit().
	a depth of 0 returns the virtual[c] unit cost function variable.
	a depth of 1 returns a CES function with virtual variables.
cost_function('sector'; depth = -1) *cost_function('sector', 'nest'; depth = -1)*	Returns the quantity times the unit cost function for the given sector and nest.
revenue_function('sector'; depth = -1) *revenue_function('sector', 'nest'; depth = -1)*	Parallel to cost_function for outputs. Note that revenue_function also builds from unit_cost_function, there is no unit_revenue_function.

(Continued)

Function	*Description*
zero_profit('sector'; depth = -1)	Returns the zero profit constraint equation for that sector.
market_clearance('commodity'; depth = -1)	Returns the market clearance constraint equation for that commodity.
income_balance('consumer'; depth = -1)	Returns the income balance constraint equation for that consumer.
compensated_demand('sector', 'commodity'; depth = -1) *compensated_demand('sector', 'commodity', 'nest'; depth = -1)*	Returns the associated demand equation for the commodity by that sector, or for supply of an output commodity.

Notes: [a] When multiple models are defined in memory at the same time, the output from a specific model can be returned by specifying the variable as an element of the model. e.g. production('model'['sector']) with the sector name as a Symbol, or production('model'['sector']['index']) for an indexed sector.

[b] The margin value is the solution value for the constraint equation associated with that variable. In a balance general equilibrium setting, these should all be approximately zero. Legitimate model solutions can include variables with non-zero margins related to complementarity, and to the more flexible option of auxiliary variables.

[c] MPSGE.jl internally generates "virtual" variables to represent unit cost functions which appear repeatedly across constraints, thereby reducing the number and size of equations passed to JuMP and improving computational efficiency, as well as enhancing readability.

Source: Authors

Any function and result can also be used as part of other functions and integrated into extended workflows[30].

Internal functions that are part of the package/module but which are not explicitly defined for users can also be called by articulating MPSGE.'function_name()'. For example, MPSGE.tax_revenue('sector','consumer') will return the equation that determines the total tax revenue from all taxes imposed on that sector going to the consumer agent. The entire open-source code defining the package is accessible online via GitHub, allowing anyone to view, download, and explore the code and its internal functions.

Elements for different models can be defined at any point because the model name appears as part of the definition[31]. Any part of a specific model or model's

[30] Meaning that any output of an MPSGE.jl model can easily be used as an input into other functionality. As a simple example, a model can be updated and solved many times in a loop, each time extracting a specific result to generate a plot. Or a workflow might use an output from a model as an input into another type of function, then use a result from that function to update the MPSGE.jl model in the next iteration.
[31] For example, production blocks for two separate models can be defined sequentially in MPSGE.jl. By contrast, MPSGE/GAMS models can only be defined in their entirety

results can be called at any point by specifying the model name with the appropriate element of the model indexed as the argument for the function.

In MPSGE.jl, model results can be easily filtered and sorted by variable, value, or margin value from the results dataframe, which is useful for troubleshooting.

In MPSGE.jl, model variables and expressions can be used in quantity fields and taxes, providing additional flexibility that is not possible with MPSGE/GAMS. The 4.2 Flexibility in MPSGE.jl: Migration Model Example highlights this feature.

Many aspects of the program 'under the hood' have been designed for efficiency and ease of use, and we highlight just two important ones here. We have implemented automatic pruning as the model is being built, allowing equations to be constructed over full index sets[32] even when many elements are irrelevant or zero. Variables, equations, and subexpressions are dropped automatically when zero values appear. For large, indexed models in particular, there may be numerous zero values amongst the data. Automatic pruning done by MPSGE.jl reduces the number of variables and constraints passed to JuMP and to the solver, which can dramatically speed up the processing and solution. Another important speed efficiency is the automatic generation of virtual variables[33] to represent unit cost functions. Generating virtual variables greatly reduces the number and size of the equations passed to JuMP and to the solver.

3.2 MPSGE.jl Parity with MPSGE/GAMS

For completeness, we briefly list some central functionality in MPSGE.jl that parallels MPSGE/GAMS[34] or GAMS.

Indexing is straight-forward for any model variable, for any number of dimensions. Inputs or outputs within a production demand block can be indexed. Sets of production blocks with indexed sectors or any other model element can be built or called using for-loops, which are computationally efficient in Julia[35]. Inputs and outputs can also be nested with nest signifiers, with the relationships between nest layers specified in the elasticity array section of the production block. The larger dataset example in Appendix D shows the syntax for this indexing, looping, and nesting functionality.

---

within individual $ONTEXT blocks.

[32] When constructing a block, it is convenient to iterate over all elements in parallel for different sectors, commodities, consumers or final demands. Automatic pruning allows for the user to iterate without pre-determining what elements may be missing or zero while avoiding the generation of unnecessary equations.

[33] Please see the note c in Table 2.

[34] MPSGE/GAMS documentation is a useful reference for further reading also relevant to this package. (https://www.gams.com/latest/docs/UG_MPSGE.html)

[35] For-loops are generally less efficient in other common general-use scientific programming languages such as R and Python.

The elasticity, price, or quantity can be set with constants (or arrays of constants for indexed versions). They can also be set as model parameters, allowing a user to update their value and re-solve the model in counterfactual simulations, as shown for a sensitivity analysis in the 4.1 Integrated Workflow Example below.

A zero-iteration limit can be set to check the benchmark, the default solve tolerance can be altered, reports of results can be generated.

3.3 Julia Advantages

Some advantages arise naturally from MPSGE.jl being embedded in Julia, and we highlight just a few here.

Julia and all registered Julia packages are open source. That means that it is completely free to use MPSGE.jl, Julia, and any registered Julia package. Open-source also means that the code underlying the Julia base and all packages including MPSGE.jl are freely available to look at. That also facilitates integrated troubleshooting through access to error and help messages from all dependencies, including the Julia base code. The open-source infrastructure also allows any user to contribute to the development of packages, or the base Julia code itself. Julia connects naturally to useful workflow tools like version control, and programming tools such as code testing and debugging.

The free and open-source foundation of the Julia language encourages an active user and developer community on discussion sites, and there are numerous free Julia tutorials online. The open-source culture and tools lead naturally to open-science practices with increased transparency, reproducibility, and collaboration for research. All Julia code used for research can easily be publicly available and hosted online, and all code and versions of any Julia packages that were used are accessible.

Additional convenience tools have been contributed by the user community to streamline higher level Julia programming needs. For example, Documenter.jl and Literate.jl automate aspects of documentation for many Julia packages. In combination with the Julia style guide, this facilitates some helpful standardization for documentation without an inordinate burden to developers.

Julia's syntax is designed to be intuitive, especially for users familiar with R, Python, or Matlab, following a mathematical structure. At the same time, Julia has been designed at its core to be computationally efficient, for example using multiple dispatch, dynamic typing, and support for parallel computing.

Julia is a general-purpose scientific programming language. Beyond the core base Julia code, there is a vast ecosystem of thousands of Julia packages, enabling extensive functionality. Users can seamlessly connect any element of the CGE modelling process to general Julia programming functions. As an example of the benefits of embedding within Julia, a user can reference and generate data in any number of formats including data frames, named arrays, the JuMP-specific DenseAxisArray containers and many others, with all the standard operations like

splicing, sorting, and filtering etc. It is also simple to upload from and write to csv, xlsx, json, the Julia-specific JLD2 type, and many other formats. Julia also has integration with notebooks including Jupyter, and other formats that incorporate markdown.

A number of structural aspects of Julia programming can be useful. For example, an entire CGE model can be wrapped as a function and called within other workflows. Function calls can be especially computationally efficient in Julia. The model function can incorporate model parameters as arguments, making it easy and streamlined to run multiple counterfactual simulations.

The "." dot operator broadcasts a function in Julia, neatly transforming scalar or individual element functions into array functions. For example, given an indexed variable U, "value.(U)" will return all indexes and their corresponding values.

Julia's macro metaprogramming, signified by the "@" symbol, provides an additional layer of extensibility. Macros operate on expressions before runtime, and can be used to generate new code from within a program. We use the flexibility of macros to facilitate a neater user interface[36] on top of the more complicated underlying code.

Julia is case sensitive and uses all Unicode symbols[37], including Greek letters and emojis, expanding the options for variable naming, and for algebraic symbols and notation in model equations.

3.4 Linking to JuMP

A number of JuMP functions have been directly ported into MPSGE.jl for convenience. For example, the function set_silent('model') suppresses the print of the solver output during and after the model solution[38]. Calls directly to JuMP can also be made with JuMP.'function()' for any function that may not have been already incorporated into the MPSGE.jl package, as long as JuMP.jl is activated with a using or import statement.

4. Example Applications

4.1 Integrated Workflow Example

As a simple illustration of the workflow benefits of embedding MPSGE.jl's CGE functionality within a general-use scientific programming language, we combine

[36] A supplementary API interface is in development to facilitate a copy and paste of MPSGE/GAMS code directly for use in MPSGE.jl

[37] https://docs.julialang.org/en/v1/manual/unicode-input/ provides a comprehensive list, along with the keyboard shortcuts to create them. Some symbols function as operators (e.g. ∈) or mathematical constants (e.g. π)

[38] This can be especially useful for running a sizeable model in a loop because printing is slow.

a CGE model with a small selection of readily available and easy-to-use Julia packages[39] to implement a Monte Carlo sensitivity analysis.

For this illustration, we generated an MPSGE.jl version of the WiNDC.jl national U.S. model[40]. The canonical WiNDC national model closely follows BEA data and uses generic values for elasticities applied uniformly across all sectors, listed in the notes below. We set the nine elasticities as model parameters and ran the sample counterfactual of removing production taxes and tariffs. Then we varied the values of each elasticity parameter by randomly sampling from probability distribution functions and re-solving the model counterfactual hundreds of times in a loop. Finally, we collected all the results in a dataframe and visualized them with scatter plots.

Figure 1 presents a subset of the sensitivity analysis results. We show the activity levels for three sectors in intermediate domestic production, along with the foreign exchange price. The variables are normalized to one in the benchmark equilibrium. In each sub-plot, the baseline counterfactual value corresponds to the solution under the generic elasticities. Each plot illustrates how the counterfactual solution would change given alternate elasticity parameter values in the model. A horizontal line indicates that the variable's solution value is unaffected by changes to that elasticity parameter.

4.2 Flexibility in MPSGE.jl: Migration Model Example

Here we show an example of additional flexibility that is available with MPSGE.jl that is not possible with MPSGE in GAMS. MPSGE.jl allows any variable or expression in the quantity field, whereas MPSGE/GAMS is limited to values and equations.

To illustrate, we construct a model representing internal migration in response to regional economic shifts. Internal migration can be significant, with important policy implications; China is a notable example. Studies report 230 million "floating population" in China in 2011, compared to UN estimates of 214 million total international migrants in the same year (Chan and Yang 2020). In our example model, we use generic regions and randomly generated values, but this framework could be expanded and combined with real data for policy-relevant analysis.

We model that a portion of the population will migrate to areas with higher utility after a productivity shock. At the same time, we want to incorporate a countervailing impetus, the negative congestion spillover from increased population density.

---

[39] Distributions.jl, Dates.jl, DataFrames.jl, Plots.jl, Tables.jl, and CSV.jl

[40] https://github.com/uw-windc is the GitHub repository for the WiNDC project. Multiple models and model versions are available from there, including the Julia version of the national model.

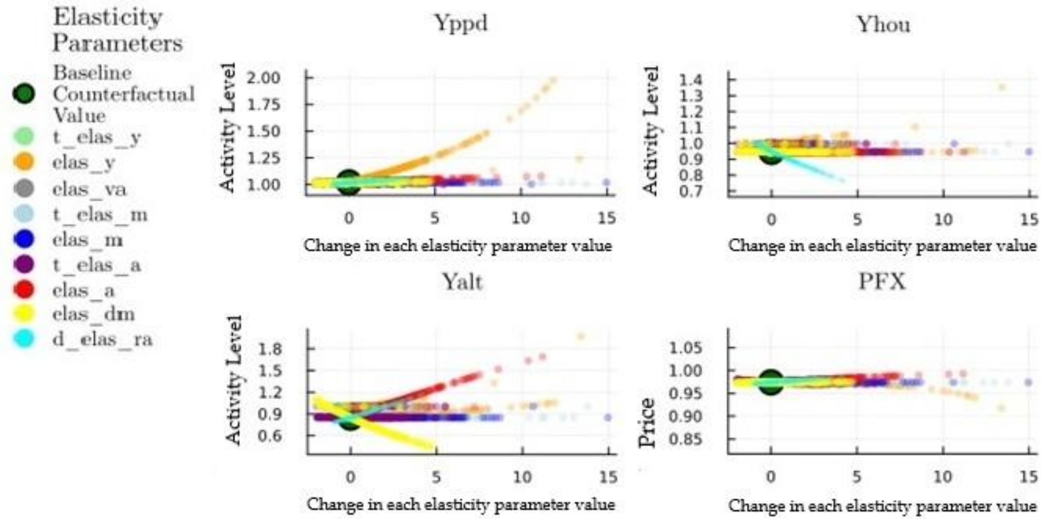

Figure 1. A sample of variables from the WiNDC national model, illustrating the sensitivity of the counterfactual results to different elasticity parameter values.

Notes:

Yppd	Domestic intermediate production for the paper products sector
Yhou	Domestic intermediate provision of housing services
Yalt	Apparel and leather and allied products
PDX	Price of foreign exchange

Parameter	Base value	Parameter description
t_elas_y	0	Elasticity of transformation for intermediate supply
elas_y	0	Elasticity of substitution for intermediate demand
elas_va	1	Elasticity of transformation for intermediate supply, value added nest
t_elas_m	0	Elasticity of transformation for margin supply
elas_m	0	Elasticity of substitution for margin demand
t_elas_a	2	Elasticity of transformation for domestic/foreign supply
elas_a	0	Elasticity of substitution for goods/margin composite
elas_dm	2	Elasticity of substitution for Armington demand nest
d_elas_ra	1	Demand elasticity of final demand

Source: Author calculations.

In MPSGE.jl we can incorporate the population congestion spillover directly as a model variable affecting the quantity of regional consumption, which in turn impacts utility. In order to achieve the same mitigating effect of population growth in MPSGE/GAMS, we have to use a workaround, detailed further below. The full scripts for both versions of the model appear in Appendix C and on GitHub[41].

4.2.1 Internal Migration Sorting

First, we build a model in both MPSGE.jl and MPSGE/GAMS where the population sorts in response to changes in local consumption from a random productivity shock. Here, utility is only a function of consumption, without the negative population congestion spillover. We will add that in a subsequent step.

To build the migration sorting model, we construct n regions, each randomly endowed with a baseline productivity of land, capital, and labor. The productivity in each region sets its benchmark income and population share. We allow capital to be mobile across regions with a single price, but the price of labor and land are regional. Each region consumes from the output of all regions, proportional to its income share.

To establish the primary impetus for the population to respond to regional shocks, we generate an index variable to track per capita consumption utility for each region.

$$U_r = \frac{M_r}{p_r M_r^0 N_r} \tag{1}$$

Where M_r is the nominal[42] total income in each region, p_r is the local price of consumption, M_r^0 is consumption at the benchmark, and N_r is the population index.

Population in each region is then determined by a logit model incorporating the per capita consumption utility index.

$$N_r = \frac{e^{\mu} U_r}{\sum \theta_r e^{\mu} U_r} \tag{2}$$

Where μ is the logit elasticity parameter, affecting the strength of the migration effect, and θ_r is the region's benchmark share of consumption. Initial population is sorted by the benchmark consumption, so the benchmark utility index and benchmark population index in each region equal 1.

As always, we first check that the model is balanced at the benchmark with a call to the solver with zero iterations[43]. Then we implement the effect of heterogenous regional economic changes, first without the congestion spillover.

[41] https://github.com/julia-mpsge/spillover_model.jl has the script of this example model.

[42] That is, the value at current prices in the model.

[43] In this case we have constructed the model to be balanced with the initial randomly generated data.

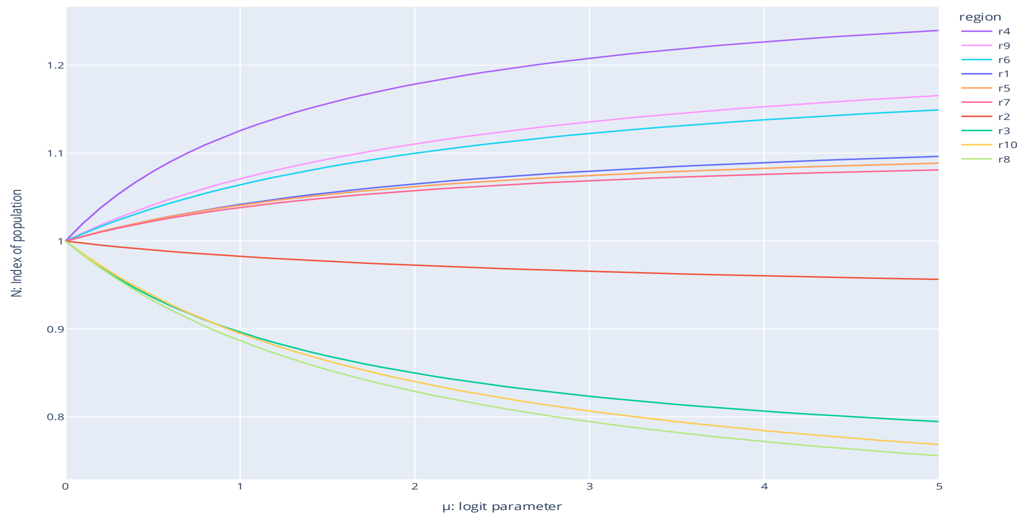

Figure 2. The strength of the sorting effect from increasing values for the parameter μ.

Source: Author calculations.

We simulate regional economic changes by imposing a set of productivity shocks. The model includes productivity parameters as coefficients on the quantity of output for each region, initially set to 1. We update those with random generated values between 0.5 and 1.5.

The cost of local production in the model adjusts to the shock. Negative shocks to local productivity increase the supply price of local goods, while positive shocks decrease it. These local price changes affect local consumption and utility in each region. Population then adjusts according to the per capita utility index (as in equation 2) such that the local consumption price is equalized in all regions, responding to the productivity changes.

Figure 2 shows the results from a model structured as described above, with 10 regions and randomly generated benchmark data. We illustrate the productivity shock simulation with a sensitivity test, showing how increasing the logit elasticity parameter, μ, impacts the migration effect. After simulating the random shocks, we re-solve the model 50 times, incrementally increasing the value for μ from 0 up to 5 in steps of 0.1, and plot the resulting population index value for each region.

In the first solve, with the logit parameter μ set to 0, population is not responsive at all to the change in local productivity, and the population index for all regions equals 1. As the value of the migration elasticity parameter increases, the migration response to the regional productivity changes becomes stronger.

4.2.2 Congestion Externality: MPSGE.jl

Now we want to incorporate the disutility of congestion as population rises in different regions. We construct a congestion externality variable that increases exponentially with the change in the population index N_r. The γ elasticity parameter determines the extent of disutility from changes in population.

$$CongExt_r = N_r^{\gamma} - 1, \;\; \gamma \geq 0 \tag{3}$$

In MPSGE.jl we are able to use the complement of the congestion externality variable $(1 - ConExt_r)$ as a coefficient on consumption in each region. This will directly affect consumption quantity, consumption utility, and therefore mitigate the migration effect as the population index N changes in each region.

Figure 3 shows how the population response changes with the congestion spillover incorporated in the model. To illustrate how the results are different with this element, we perform a sensitivity test showing how increasing values for the γ elasticity parameter alters the migration effect. With the same random productivity shocks as above and the μ parameter value on the population index fixed at 5, we run the model in a loop again, this time gradually increasing the value for γ from 0 up to 5.

In this case, in the first solve with the population density externality elasticity parameter γ value equal to 0, the response is exactly equivalent to the migration effect at μ=5 as in Figure 2. Population responds to the productivity shock, not affected by the congestion externality. As the value of the γ parameter increases, the migration response is dampened by the negative population density spillover.

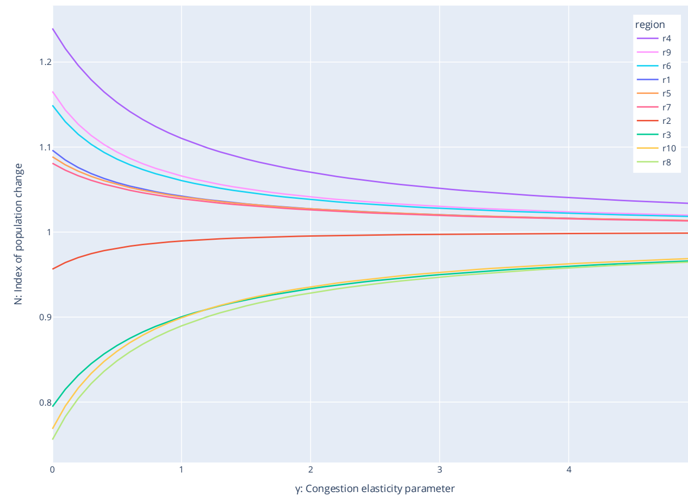

Figure 3. Altering γ, the responsiveness of migration to the negative effect of increased population.

Source: Author calculations.

4.2.3 Congestion Externality: MPSGE/GAMS

For the initial internal migration sorting section of the model, both the MPSGE.jl and the MPSGE/GAMS versions are parallel. However, it is not possible to include a variable in the quantity field of a production block in MPSGE/GAMS as we have done above. To create the same effect, we need to set the congestion externality as an artificial endogenous tax on consumption, in conjunction with a rationing instrument negatively balancing the tax revenues in the endowment.

The endogenous tax on consumption is defined in the same way as $CongExt_r$ in equation 3 above. As a tax, this becomes a complement coefficient on consumption $(1 - CongExt_r)$ as above. However, with $CongExt_r$ as a tax, we now have revenue going to the representative agent. To counter the revenue distribution from the tax, we must set a negative endowment as a function of the congestion spillover.

$$Rationing_r = C_r \cdot consumption_r^0 \cdot CongExt_r \tag{4}$$

Where C_r is the index of consumption in region r, $consumption_r^0$ is the benchmark consumption level for region r, and $CongExt_r$ becomes a congestion 'tax' on consumption. Set with a negative one quantity and the price of local production, $Rationing_r$ will track the externality 'tax' revenue and subtract it from the representative agent's budget.

With this example, we show that while it is possible to work around the restriction in MPSGE/GAMS, the additional flexibility in the Julia version allows for a more compact and intuitive structure for this question. With MPSGE.jl, any model element, including values, equations, variables, parameters, or combinations of these elements as part of expressions can all be used in the quantity field[44] of a production block.

4.3 Large Dataset Example: U.S. Tariff Effect on Households in each State

Finally, to illustrate the efficacy of the package at scale, we build a model with a large number of variables and constraint equations, and a large dataset. We replicate the canonical WiNDC[45] U.S. regional model (T. Rutherford and Schreiber 2019), disaggregated into the 50 U.S. states plus D.C., with representative households in each region split into 5 income quintiles.

We will use this model to evaluate the effect of U.S. tariff increases. The impact of tariffs in the U.S. continues to be a salient economic question. The dynamic nature of U.S. tariff policy makes the ability to provide efficient analysis especially

[44] Or tax or elasticity fields.

[45] The Wisconsin, National Data Consortium (WiNDC) generates free and open-source, balanced, micro-consistent databases from public data, suited for CGE modelling. https://windc.wisc.edu/. They provide canonical models that link directly to the balanced datasets as a foundation for adaptation for specific research questions.

useful. Theory suggests that tariffs raise prices for consumers, though the magnitude and distribution can be hard to predict. We test the impact on consumers differentiated by region and household income levels.

We employ a generic example, setting a uniform tariff floor without explicit bilateral trade or tariffs. Our intention here is to demonstrate the relative ease of using the package for policy-relevant questions, and to provide an accessible, foundational framework. We note that even without the bilateral tariff details, the impact on households differentiated by both state and income provides some insight.

We briefly describe the model here[46], and present the results from a simulation that increases tariffs on all imported goods to a minimum of 10%. We also use this model and policy simulation to demonstrate the speed advantages of the package. A full transcript of the model is included in Appendix D[47].

The WiNDC regional model with disaggregated households contains 11,478 sectors, 15,223 commodities, and 258 consumers. Each of the 50 U.S. states and the District of Columbia produce and supply goods into separate local, national, and foreign markets. Production in each state is made up of 71 sectors, with each sector producing a single representative commodity. Each sector in each state combines labor and capital with local, national and imported goods from the same 71 sector categories for production. Trade and transport margins are paid on each good produced in each state.

Households at each income level provide labor and capital, receive transfers, and maximize utility from leisure and from the consumption of goods and services within the same 71 categories. Labor is mobile across sectors but fixed within states and levels of income. Capital is mobile. The government receives all taxes which it uses to purchase goods and services or passes on to households as transfers. An endogenous level of transfers ensures the balance of the government budget. Existing taxes are represented within the following categories: a uniform capital tax; import taxes/tariffs, production taxes, and consumption taxes (each differentiated by sector); medical and social security (uniform up to household income level 3, but differentiated by state and household level for income levels 4 and 5); and marginal labor taxes (differentiated by household income level per state).

A portion of output from each sector in each state goes to investment, at a level adjusted endogenously by an economy-wide savings rate. The savings rate is a

[46] A full technical description of the routine for preparing all the associated data is detailed in (Rutherford and Schreiber 2019).

[47] We note that running a model of this size requires a PATH solver license, free for non-commercial use. We include a reference for obtaining and using the license in the README on the MPSGE.jl GitHub repository available at https://github.com/julia-mpsge/MPSGE.jl .

function of the value of capital stock, which is a function of the return rate on capital. The foreign exchange rate is fixed.

We represent this version of the U.S. economy for 2021 with a set of production and demand blocks, linked to the economic data that has been calibrated to produce the necessary mathematical balancing. The equations for the equilibrium conditions are automatically generated by the program. To confirm that the model and data are balanced, we call the solver to assess the full set of model equations with the benchmark variable values by restricting the number of solving iterations to zero. The program builds the full model, passes it to PATH via JuMP.jl and PATHSolver.jl, and PATH evaluates the full system of constraints at their benchmark. PATH returns the postsolved residual, which should be approximately zero, that is less than the tolerance level, e.g. 10^{-6} in absolute value.

To run our tariff simulation, we only need to update the tariff parameter from their benchmark values with a set_value statement[48] and re-solve the model, this time removing the iteration limit. PATH algorithmically determines the full set of endogenous variable values that simultaneously satisfy all constraints to less than or equal to the default tolerance.

In Figure 4, we show how the 5 household levels in each state are affected by the rise in tariffs. The plot is sorted from left to right by the effect on the lowest income quintile in each region. All households in all regions experience a negative shock to their overall real consumption. In most states, the effect on consumption is reduced with increasing income, showing a regressive pattern. There is some variation, however. For example, the effect is progressive up to the second highest income group in both Utah and Idaho.

It is not easy to predict how tariffs might affect different regions without a detailed general equilibrium analysis. Figure 5 illustrates that the impacts go beyond a simple correlation with import exposure. The plot shows net imports for each region from the benchmark data, weighted by each region's gross state product. With the states ordered in parallel, comparing Figure 5 with Figure 4 shows that the real consumption impact does not align with the size of net imports as we might intuitively expect. In fact, the largest impacts occur in the two states that are the biggest net exporters.

We also calculate an overall national Consumer Price Index (CPI), tracking the average change in prices faced by consumers as a weighted sum of the basket of goods purchased by each income level household in each state. In our central policy simulation, adding a tariff floor of 10%, the weighted CPI increases by 2.7%.

---

[48] The tariff parameter is a matrix with benchmark values for each representative commodity determined from the data, equal for each state. We update all values within a small for-loop, with an if statement to exclude any values already greater than our 10% floor.

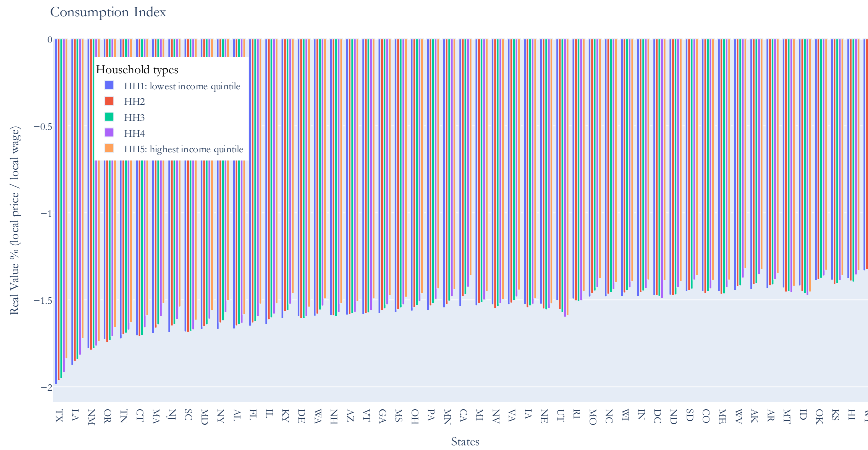

Figure 4. Change in real consumption for all household income levels in each state as the result of increasing all tariffs to a floor of 10%.

Source: Author calculations.

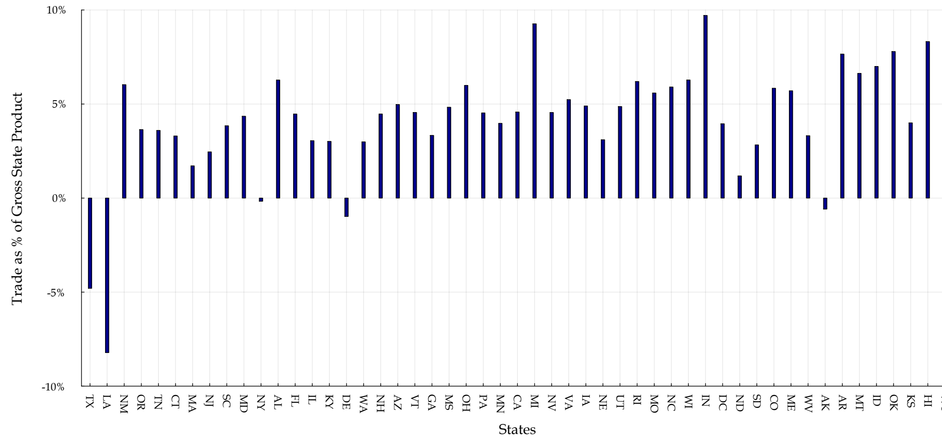

Figure 5. Net imports as a percentage of each region's gross state product.

Source: Author calculations.

We examine the relationship between the tariff floor levels and the CPI by iterating over tariffs, gradually increasing rates. We plot the overall CPI change as a function of the tariff floor rate in Figure 6.

We can also look at the effect on the prices of specific commodities in each region. To illustrate the heterogeneity across states, we focus on a sample of three commodities. Figure 7 shows that the tariff floor leads to a fairly consistent 3-4% increase in prices for agricultural products across all regions, a slightly smaller and mostly consistent increase in food and beverage products, and price increases for petroleum products that are more variable in magnitude.

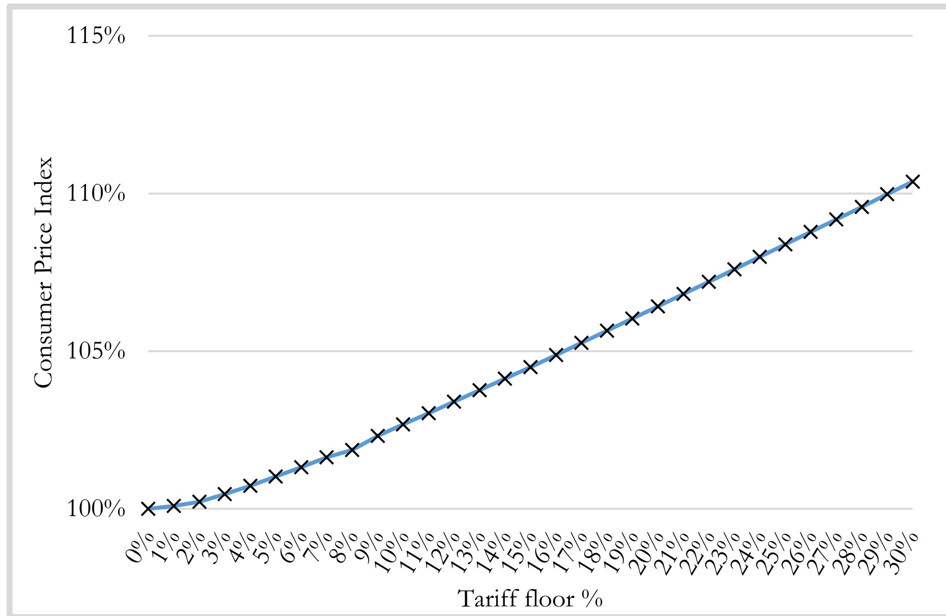

Figure 6. Change in the Consumer Price Index as a function of the tariff floor rate.

Source: Author calculations.

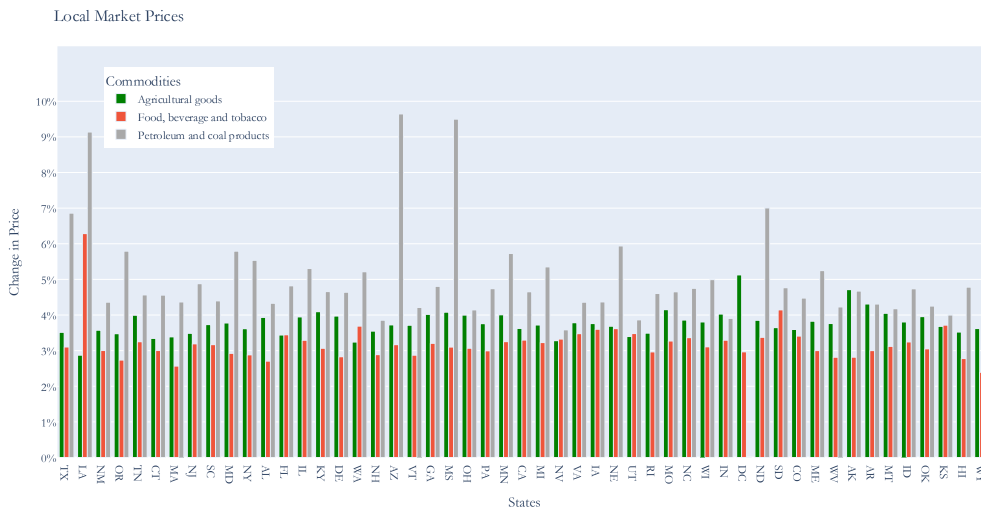

Figure 7. Local price changes for a sample set of commodities in each state as the result of increasing all tariffs to a floor of 10%.

Source: Author calculations.

4.3.1 Benchmark Test

Here we present a computational speed profile of the package, using the same WiNDC U.S. regional model with disaggregated households as above, and the same counterfactual simulation. The MPSGE.jl package leverages Julia as a high-performance programming language that can be as fast as C or Fortran, but with Julia's intuitive syntax that is similar to Python or R. We compare that to parallel computational speeds[49] from the MPSGE/GAMS version of the same model.

To demonstrate the speed comparison, we separately benchmark four stages of running a CGE model:

1) Loading the data – Both datasets are stored in formats optimized for the language. GAMS loads a GDX file and the Julia version loads a JLD2 file.
2) Building the Model – MPSGE.jl explicitly builds the model equations whereas MPSGE/GAMS writes a `GEN` file which is used to generate the model.
3) Verify the Benchmark – This step verifies that the input data is calibrated; it sets the PATH iteration limit to 0 which effectively evaluates the constraints at the variable starting values.
4) Solve a Counterfactual – We set a uniform 20% import duty across all regions and goods. This is solved with the default PATH options and an iteration limit of 10,000.

For the Julia results, we used the standard built-in `@benchmark` macro from the `BenchmarkTools` package. Each stage was measured independently, specifying that 10 runs were to be executed for each. The @benchmark` macro automatically disregards the initial compilation time from the first run of a Julia object, since this is not representative of the code's execution time.

For the MPSGE/GAMS version, we created a sequence of files to isolate each step: the first file loads the data; the second loads the data and builds the model, and so on. The times were then adjusted to capture only the additional stage introduced in that file. GAMS has no built-in benchmarking tool. The execution time of each file was measured using the `Measure-Command` PowerShell command. Each reported time is an average of 10 runs. Table 3 below benchmarks MPSGE.jl alongside MPSGE/GAMS.

[49] These benchmark results were run in Windows 11 on a machine with an i7-9700k processor and 32GB of RAM.

Table 2. Speed comparison between MPSGE.jl and MPSGE/GAMS in seconds.

	MPSGE.jl	MPSGE/GAMS
Load Data	0.02	0.06
Build Model	23.71	0.53
Verify Benchmark	2.27	194.85
Solve Counterfactual	24.61	668.39
Total Time (seconds)	50	863

Source: Author calculations.

5. Discussion

This paper presents the current version of the MPSGE.jl package. We have demonstrated the fundamentals of using the package, highlighted some of its benefits, and given a sense of the package's potential. While the package is currently fully functional and available for use, the capacity for further development is an additional key feature of the open-source nature of the package. As an open-source Julia package hosted on GitHub, it is easy and accessible to contribute collaboratively to the ongoing development of MPSGE.jl. Any interested user can suggest a new feature, improvement, or report a bug as an 'issue' in the GitHub repository. Anyone with sufficient interest and skill can make changes to a local copy of the code, test it, and offer those changes as a fully functional update to the package itself.

We have a number of features proposed or in development, and we mention a handful here. One feature in development is an interface that could allow GAMS/MPSGE code to be copied directly into Julia and run with MPSGE.jl. This feature could facilitate experimentation and easier transition for users with existing background and/or code in GAMS.

Integrated data preparation can streamline the workflow for CGE modelling, particularly by enabling seamless selection of different data years or updating models as newer data becomes available. There is a natural alignment with open-source CGE data, as used in the U.S. tariff simulation example model described above. WiNDC builds balanced SAM datasets for the U.S. from public data. WiNDC.jl is a Julia implementation of the WiNDC CGE data project. An additional feature could integrate the connection with WiNDC.jl directly into the MPSGE.jl package. A function could facilitate single-click data download and calibration feeding directly into an MPSGE.jl model. Furthermore, while WiNDC and WiNDC.jl currently generate CGE-ready data for the U.S., the template could be used to expand to other countries and regions, further advancing accessible CGE modelling and research.

The GTAP project[50] includes the most commonly used multi-region global model (Hertel 1998; Corong et al. 2017) in CGE research. We currently have an initial working version of the GTAP model v9 in MPSGE.jl. More comprehensive integration and implementation of the GTAP models and GTAP data is another development priority.

The Constant Elasticity of Substitution (CES) and Constant Elasticity of Transformation (CET) functional forms provide significant flexibility for characterizing the important relationships in a CGE model. However, other functional forms may better represent economic relationships for particular models. There is potential to incorporate other functional forms for the constraints as another direction for future development.

MPSGE.jl has been developed for modelling with any script editor, notebook, programming interface, or integrated development environment such as VS Code. MPSGE.jl could also form the basis of a web- or browser-based interface and bypass the need to run or understand code entirely. Users could make selections from dropdowns and other customization options to generate and run models. This type of interface could support models suitable for research, or be particularly useful for pedagogical purposes.

It is straightforward to output the constraint equations generated by MPSGE.jl, however, the full versions of the equations can become long, imposing, and difficult to parse for large models. While there are currently functions to print more succinct versions of the equations with sub-expressions as variables, we hope to add the ability to print equations in the form that would appear in an academic paper, with indexing. An automatic translation from the models as defined in MPSGE.jl to equations as they would be represented in text could be a great convenience.

6. Conclusion

We have given an overview of the Mathematical Programming System for General Equilibrium in Julia. MPSGE.jl provides succinct-form CGE model development in a platform that offers a number of novel features and advantages.

Speed and functionality of the MPSGE.jl package are at least comparable to the alternatives. The package extends the benefits of the more human-readable, mathematically-based intuitive syntax of the Julia language, which was intentionally developed to provide high performance in combination with ease of use (Bezanson et al. 2017). The open-source architecture provides greater access, transparency, integration and usability, as well as collaborative and development potential.

[50] More details are available at https://www.gtap.agecon.purdue.edu/default.asp.

MPSGE.jl does not require any paid license[51]. The package is hosted publicly on GitHub, offering all the standard open-source developments workflow functionality such as version control, issue tracking, contributor management, and all the transparency, collaboration, and reproducibility advantages. Use of Julia is free and integrates well with various free general programming IDEs, such as Visual Studio code and Jupyter notebooks, and an array of code development infrastructure, from the VS Code Julia extension to documentation, package registration and release tagging, maintenance etc.

MPSGE.jl, the Julia programming language, and the vast majority of more than 10,000 registered packages[52] are open-source and hosted on GitHub. Any user can look at all of the code, including the entire history, debug deeply through all package dependencies and the code stack, and connect to documentation, error messages, and other peripherals, all in the same language. Any user can contribute feature ideas, submit bug reports, or directly make improvements and enhancements to the code and package itself. Users working collaboratively can jointly interact with our source code, and more easily and seamlessly incorporate collaborative workflows and research transparency. We consider this work to be a part of the open-science movement, improving science robustness, credibility, collaboration, communication, and results.

Acknowledgements

We thank the editors and anonymous reviewers for helpful suggestions. Financial support for Eli Lazarus from the Precourt Energy Institute at Stanford University is gratefully acknowledged.

References

Bezanson, J., A. Edelman, S. Karpinski, and V. B. Shah. 2017. "Julia: A Fresh Approach to Numerical Computing." *SIAM Review* 59 (1): 65–98. https://doi.org/10.1137/141000671.

Chan, K. W., and X. Yang. 2020. "Internal Migration and Development: A Perspective from China." In *Routledge Handbook of Migration and Development*, edited by T. Bastia and R. Skeldon. New York: Routledge.

[51] MPSGE.jl uses the PATH solver. The use of PATH requires a license, but it is free, available via the https://pages.cs.wisc.edu/~ferris/path/. MPSGE.jl links to PATH via the Julia wrapper package PATHSolver.jl

[52] Julia also provides simple methods for connecting to the libraries of other programming languages, facilitating the development of fairly simple Julia wrappers to access existing functionality in other languages until and unless parallel packages are available natively.

Corong, E., H. Thomas, M. Robert, M. Tsigas, and D. Van der Mensbrugghe. 2017. "The Standard GTAP Model, Version 7." *Journal of Global Economic Analysis* 2 (1): 1–119. https://doi.org/10.21642/JGEA.020101AF.

Dunning, I., J. Huchette, and M. Lubin. 2017. "JuMP: A Modeling Language for Mathematical Optimization." *SIAM Review* 59 (2): 295–320. https://doi.org/10.1137/15M1020575.

Hertel, T. W., ed. 1998. *Global Trade Analysis: Modeling and Applications*. Cambridge: Cambridge University Press.

Lange, O. 1942. "Say's Law: A Restatement and Criticism." In *Studies in Mathematical Economics and Econometrics*, edited by O. Lange, F. McIntyre, and T. O. Yntema. Chicago: University of Chicago Press.

Lubin, M., O. Dowson, J. D. Garcia, J. Huchette, B. Legat, and J. P. Vielma. 2023. "JuMP 1.0: Recent Improvements to a Modeling Language for Mathematical Optimization." *Mathematical Programming Computation* 15 (3): 581–89. https://doi.org/10.1007/s12532-023-00239-3.

Mathiesen, L. 1985. "Computational Experience in Solving Equilibrium Models by a Sequence of Linear Complementarity Problems." *Operations Research* 33 (6): 1225–50. https://doi.org/10.1287/opre.33.6.1225.

Meeraus, A. 1983. "An Algebraic Approach to Modeling." *Journal of Economic Dynamics and Control* 5: 81–108. https://doi.org/10.1016/0165-1889(83)90016-7.

Rutherford, T. F. 1999. "Applied General Equilibrium Modeling with MPSGE as a GAMS Subsystem: An Overview of the Modeling Framework and Syntax." *C omputational Economics* 14: 1–46. https://doi.org/10.1023/A:1008655831209.

Rutherford, T. F. 1987. "Applied General Equilibrium Modeling." Ph.D. thesis, Department of Operations Research, Stanford University, Stanford, the United States of America.

Rutherford, T., and A. Schreiber. 2019. "Tools for Open Source, Subnational CGE Modeling with an Illustrative Analysis of Carbon Leakage." *Journal of Global Economic Analysis* 4 (2): 1–66. https://doi.org/10.21642/JGEA.040201AF.

Scarf, H. E., and T. Hansen. 1973. *The Computation of Economic Equilibria*. Monograph (Yale University. Cowles Foundation for Research in Economics); 24. Yale University Press. WorldCat.

Appendix A. Structure and Details of the MPSGE.jl Package

At its core, the MPSGE.jl package creates a JuMP.jl model, structured as a mixed complementarity problem (mcp). A mixed complementarity problem links a variable to each equation such that either the equation is satisfied[53] or the variable is its bound, for example zero.

A.1 MPSGE Variables

There are five types of variables in MPSGE: parameters, sectors, commodities, consumers, and auxiliary variables.

Parameters are fixed variables that can be modified between solves. These are used to introduce exogenous values into the model; a common example being a tax. MPSGE.jl requires parameters to be explicitly defined in a `@parameter` macro. This is in contrast to GAMS where all non-variable values in a model are considered to be parameters.

Sectors, or activities, are production activities that convert commodity inputs into commodity outputs. The variable associated with a sector is the **activity level** of that sector. Sectors are assumed to be initially operating at 100% capacity, which makes the starting level be 1. Each sector corresponds to exactly one production block in the model.

Commodities, or markets, are goods or services that are bought and sold in the economy. The variable associated with a commodity represents the **price** of the commodity. Initial prices are assumed to be 1. We will discuss the relationship between the price, activity level, and quantity when we discuss production blocks.

Consumers are representative agents with final demands and they supply market with endowments, receive tax revenue, and pay subsidies. The variable associated with a consumer is the total **income** of the consumer with starting value equal to the initial income level.

Auxiliary variables are used to represent additional economic concepts not captured by sectors, commodities, or consumers. They can represent endogenous quantities which are functions of other variables such as prices and quantities. Auxiliary variables have a default start value of `0` and are unbounded unless otherwise specified.

[53] For MPSGE, sector activity levels are linked to the zero profit constraints, commodity prices are linked to the market clearance constraints, and auxiliary variables are linked to their constraints. If a zero profit constraint solves with equality (unit cost = unit revenue), the sector activity will generally be positive, whereas if it solves with inequality (unit cost > unit revenue) the associated activity goes to its zero lower bound, a corner solution with the intuition that the sector would not operate at a loss. (Strictly speaking, both can occur simultaneously, with a zero lower bound met exactly at equality).

A.2 Production Blocks

In MPSGE, we think of each sector as a representative firm. Firms produce output commodities by combining input commodities. The specific inputs and outputs for each sector are defined in a production block. MPSGE takes the information in a production block and generates *cost functions* (not production functions).

Production blocks define the input and output trees. Non-leaf nodes in these trees are referred to as nests and have an associated elasticity. Leaves are associated with a commodity and are required to include a reference quantity and parent nest. Leaves can optionally include a reference price and taxes.

```
@production(Model, Sector, [Nesting Structure], begin
     @output(P₁, Q₁, nest₁)
     @input(P₂, Q₂, nest₂)
end)
```

A.3 Specifying Leaves

Leaves are specified using the `@input` and `@output` macros within the `begin … end` block within the production macro. The two macros have the same structure, so we will discuss the input side. The input macro has the form:

$@input(P, Q, n, taxes\ =\ [Tax(H, tx)], reference_price\ =\ rp)$

where: P is the commodity variable, representing price; Q is the reference quantity; n is the parent of this node in the tree; taxes is an optional keyword, in this case we specify a tax with value `tx` paid to consumer `H`; reference_price is used to ensure the tax-adjusted price has an initial value of 1.

The tax-adjusted price, $\bar{P}$, for an input with taxes t_i and reference price rp is given by:

$$\bar{P} = \frac{P(1 + \sum t_i)}{rp} \tag{A.1}$$

The reference quantity, Q , is also adjusted by the reference price. The corresponding quantity is given by:

$$\bar{Q} = Q \cdot rp \tag{A.2}$$

For convenience, we will refer to the tax-adjusted price, $\bar{P}$, as the unit cost function of the leaf.

Inputs/outputs can also be indexed. For a set S we can specify an indexed input using the syntax:

$@input(P[s = S], Q[s], n)$

This will create an input for each $s \in S$ using the value $Q[s]$ as the quantity.

A.4 Specifying Nests

Nests are specified in an array in the third argument of the `@production` macro. There must be exactly two top-level nests, indicating the roots of the two trees. These are specified using the syntax:

$name = elasticity$

In MPSGE.jl the names of the top-level nests can be any Julia expression. MPSGE/GAMS requires the names to be `s` for the input tree and `t` for the output tree, representing substitution and transformation respectively. Non-root nests must also specify their parent:

$name \Rightarrow \text{parent} = elasticity$

Non-root nests can also be index. Given a set S, we can define:

$name[s = S] \Rightarrow \text{parent} = elasticity[s]$

This will create a nest for each $s \in S$.

A.5 Building Cost Functions

Each node in the tree has a corresponding quantity and unit cost function. Let N be a non-leaf node in the input tree with elasticity σ and k children, where child i has (adjusted) quantity Q_i and unit cost function C_i. The quantity of N is the sum of its children quantities, or

$$Q = \sum_i^k Q_i \tag{A.3}$$

The unit cost function, C, of N is given by:

$$C = \begin{cases} \left(\sum_i^k \frac{Q_i}{Q} \cdot C_i^{1-\sigma} \right)^{\frac{1}{1-\sigma}}, & where\ \sigma \neq 1 \\ \prod_i^k C_i^{\frac{Q_i}{Q}}, & where\ \sigma = 1 \end{cases} \tag{A.4}$$

A.6 Zero Profit

Let C and R be the unit cost functions for the input/output trees respectively. The profit of the sector is then $\Pi = R - C$ and the zero profit condition is

$$-\Pi = 0 \perp X \tag{A.5}$$

Where X is the activity level of the sector.

A.7 Demand Blocks

Demand blocks in MPSGE specify consumer final demands and endowments. A demand block is structured similarly to a production block, except there is no nesting structure.

```
@demand(Model, Consumer, begin
     @final_demand(P_1, Q_1)
     @endowment(P_2, Q_2)
end)
```

The macros `@final_demand` and `@endowment` take a commodity and a quantity. If H is the consumer variable, then the income balance equation is given by:

$$H = \sum_i E_i \cdot P_i + \sum_j t_j \cdot Q_j \cdot P_j \cdot X_j \cdot \frac{\partial(\Pi_X)}{\partial(\bar{P}_j)} \perp H \tag{A.6}$$

Where the first term gives the total value of the endowments and the second term is the tax revenue paid to the consumer. The partial derivative is given by Hotelling's lemma and represents the compensated demand.

A.8 Market Clearing Conditions

MPSGE automatically generates the market clearing conditions. For completeness, we include this condition for a commodity P. Let X_i be the sectors with P as a netput, E_j be the endowments on P, and H_k the consumers demanding P. Then the market clearing condition for P is given by:

$$\sum_i X_i \cdot \frac{\partial(\Pi_{X_i})}{\partial(\bar{P})} - \sum_j E_j + \sum_k \frac{H_k}{P} = 0 \perp P \tag{A.7}$$

A.9 Auxiliary Constraints

Finally, each auxiliary variable gets associated with a constraint. The syntax is given by:

```
@aux_constraint(Model, A, Constraint)
```

Where A is the auxiliary variable and $Constraint$ is the associated constraint, which is assumed to equal 0. This gets translated into:

$$Constraint = 0 \perp A$$

Appendix B. Basic Model, MCP Julia JuMP[54], MPSGE.jl[55] and MPSGE/GAMS[56] versions.

The model, adapted from https://www.mpsge.org/markusen/m2.htm#m2_2s, includes a basic two-good, two-factor closed economy with fixed factor endowments and one representative consumer. The model includes intermediate inputs and nesting. The initial data for this model is represented in Table B.1:

Table B.1 Social Accounting Matrix data for the basic model example.

Markets	X	Y	W	CONS
PX	120	-20	-100	
PY	-20	120	-100	
PW			200	-200
PL	-40	-60		100
PK	-60	-40		100

Source: https://julia-mpsge.github.io/MPSGE.jl/v0.6.4/Tutorials/intermediate_examples/m22/.

In this model there are three production sectors, X, Y, and W, and a single consumer CONS. The rows correspond to markets for five commodities: two goods PX and PY, a welfare good PW, and two primary factors labor, PL, and capital, PK. Positive values represent supply to the market, a sale, or an output. Negative values represent demand from the market, a purchase, or an input.

The total value of good PX purchased by sector X is $120 \cdot PX \cdot X$, where X is the activity level of sector X. We take the unit on 120 to be such that the initial activity level of sector X and price of PX are both 1 and say that 120 is the *representative quantity*[57] of good PX for sector X. This assumption greatly simplifies model verification as we are starting with a balanced dataset, meaning the row and columns sums are zero. Unital initial values means, at the initial values, all constraints simplify to either row or column sums of the SAM, which should be zero. Neither unital initial values nor balanced data are strictly necessary

[54] https://github.com/julia-mpsge/markusen_2_2.jl has both the MPSGE.jl and mcp with JuMP.jl versions as functions.
[55] https://julia-mpsge.github.io/MPSGE.jl/v0.6.4/Tutorials/intermediate_examples/m22/ also shows this model with more detailed explanation and the GAMS code in pa rallel.
[56] https://www.mpsge.org/markusen/m2.htm#m2_2s links to Markusen's original GAMS version of this model.
[57] For example, it could represent 120 apples, 120 tons of apples, 120 million boxes of apples etc.

conditions for a CGE model, but they are common assumptions in the literature and make model verification easier.

This example is a scalar model, meaning no sets or indices are used. This simplifies the model and allows us to focus on the structure of the model and the syntax of MPSGE. Table B.2 below shows the expected results from the initial model benchmark and the counterfactual simulation.

Table B.2 Benchmark and counterfactual results from the basic model example.

Variable	Benchmark	Counterfactual
X	1.000	0.861867
Y	1.000	1.10915
W	1.000	0.983827
PX	1.000	1.15116
PY	1.000	0.868686
PL	1.000	0. 84119
PK	1.000	0.787062
PW	1.000	1.000
CONS	200.000	196.765

Notes: All three versions will return the same results

Source: https://julia-mpsge.github.io/MPSGE.jl/v0.6.4/Tutorials/intermediate_examples/m22/

Listing B.1 MCP Julia JuMP Version of the Basic Model.

```
using JuMP, PATHSolver
 mcp = Model(PATHSolver.Optimizer)

    @variable(mcp, X, (start = 1))
    @variables(mcp, begin
        Y, (start = 1)
        W, (start = 1)
        PX, (start = 1)
        PY, (start = 1)
        PW, (start = 1)
        PL, (start = 1)
        PK, (start = 1)
        CONS, (start = 200)
        tax_pl in JuMP.Parameter(0)
        tax_pk in JuMP.Parameter(0)
    end)

    cost_X_va = (PL*(1+tax_pl))^(40/100) * (PK*(1+tax_pk))^(60/100)
    cost_X = (20/120 * PY^(1-.5) + 100/120 * cost_X_va^(1-.5))^(1/(1-.5))
    revenue_X = PX
```

```
    cost_Y_va = (PL)^(60/100) * (PK)^(40/100)
    cost_Y = (20/120 * PX^(1-.75) + 100/120 * cost_Y_va^(1-.75))^(1/(1-.75))
    revenue_Y = PY

    cost_W = PY^.5*PX^.5
    revenue_W = PW

        # Zero Profit
        @constraints(mcp, begin
            zp_X, 120*cost_X - 120*revenue_X ⊥ X
            zp_Y, 120*cost_Y - 120*revenue_Y ⊥ Y
            zp_W, 200*cost_W - 200*revenue_W ⊥ W
        end)

        # Market Clearance
        @constraints(mcp, begin
            mc_PX, 120*X - 20*Y*(cost_Y/PX)^.75 - 100*W*(cost_W/PX) ⊥ PX
            mc_PY, -20*X*(cost_X/PY)^.5 + 120*Y - 100*W*(cost_W/PY) ⊥ PY
            mc_PW, 200*W - CONS/W ⊥ PW
            mc_PL, -40*X*(cost_X/cost_X_va)^.5*(cost_X_va/(PL*(1+tax_pl))) -
60*Y*(cost_Y/cost_Y_va)^.75*(cost_Y_va/PL) + 100 ⊥ PL
            mc_PK, -60*X*(cost_X/cost_X_va)^.5*(cost_X_va/(PK*(1+tax_pk))) -
40*Y*(cost_Y/cost_Y_va)^.75*(cost_Y_va/PK) + 100 ⊥ PK
        end)

        # Income Balance
        @constraint(mcp, ib,
            CONS - 100*PL - 100*PK -
40*tax_pl*PL*X*(cost_X/cost_X_va)^.5*(cost_X_va/(PL*(1+tax_pl)))  -
60*tax_pk*PK*X*(cost_X/cost_X_va)^.5*(cost_X_va/(PK*(1+tax_pk))) ⊥ CONS
        )

fix(PW,1;force=true)

set_optimizer_attribute(mcp, "major_iteration_limit", 0)
optimize!(mcp)

object_dictionary(mcp)
Benchmarkresults_mcp = Dict(
    k => value.(v) for
    (k, v) in object_dictionary(mcp) #if v isa AbstractArray{VariableRef}
)
mc_PX
value(mc_PX)
all_constraints(mcp; include_variable_in_set_constraints = true)

set_parameter_value(tax_pl,.5)
set_parameter_value(tax_pk,.5)

set_optimizer_attribute(mcp, "major_iteration_limit", 10000)
optimize!(mcp)
Counter1_results = Dict(
    k => value.(v) for
    (k, v) in object_dictionary(mcp) #if v isa AbstractArray{VariableRef}
)
```

Listing B.2 The MPSGE.jl Version of the Basic Model.

```
using MPSGE
M = MPSGEModel()

@parameters(M, begin
    tax_pl, 0, (description = "Tax on labor in sector X")
    tax_pk, 0, (description = "Tax on capital in sector X")
end)

@sectors(M, begin
    X, (description = "Activity level for sector X")
    Y, (description = "Activity level for sector Y")
    W, (description = "Activity level for sector W (Hicksian welfare index)")
end)

@commodities(M, begin
    PX, (description = "Price index for commodity X")
    PY, (description = "Price index for commodity Y")
    PW, (description = "Price index for welfare (expenditure function)")
    PK, (description = "Price index for primary factor K")
    PL, (description = "Price index for primary factor L")
end)

@consumer(M, CONS, description = "Income level for consumer CONS")

@production(M, X, [t=0, s=.5, va=>s=1], begin
    @output(PX, 120, t)
    @input(PY, 20, s)
    @input(PL, 40, va, taxes=[Tax(CONS,tax_pl)])
    @input(PK, 60, va, taxes=[Tax(CONS,tax_pk)])
end)

@production(M, Y, [t=0, s=.75, va=>s=1], begin
    @output(PY, 120, t)
    @input(PX, 20, s)
    @input(PL, 60, va)
    @input(PK, 40, va)
end)

@production(M, W, [t=0, s=1], begin
    @output(PW, 200, t)
    @input(PX, 100, s)
    @input(PY, 100, s)
end)

@demand(M, CONS, begin
    @final_demand(PW, 200)
    @endowment(PL, 100)
    @endowment(PK, 100)
end)

fix(PW, 1)
solve!(M, cumulative_iteration_limit=0)
M.object_dict

M.jump_model
M
Benchmarkresults = generate_report(M)
market_clearance(PX)
value(market_clearance(M[:PX]))
value(PX)
```

```
set_value!(M[:tax_pl], 0.5)
set_value!(M[:tax_pk], 0.5)
solve!(M)
Counter1_results_M = generate_report(M)
```

Listing B.3 The MPSGE/GAMS Version of the Basic Model.

```
SCALAR  TXL     Labor income tax rate,
        TXK     Capital income tax rate;

$ONTEXT

$MODEL:M2_2

$SECTORS:
        X       ! Activity level for sector X
        Y       ! Activity level for sector Y
        W       ! Activity level for sector W (Hicksian welfare index)

$COMMODITIES:
        PX      ! Price index for commodity X
        PY      ! Price index for commodity Y
        PL      ! Price index for primary factor L
        PK      ! Price index for primary factor K
        PW      ! Price index for welfare (expenditure function)

$CONSUMERS:
        CONS    ! Income level for consumer CONS

$PROD:X s:0.5 va:1
        O:PX  Q:120
        I:PY  Q:20
        I:PL  Q:40    va:
        I:PK  Q:60    va:

$PROD:Y s:1
       O:PY     Q:100
        I:PLS    Q:60
        I:PKS   Q:40


$PROD:W s:1
       O:PW   Q:200
        I:PY     Q:100
        I:PL     Q:100

$DEMAND:CONS
        D:PW   Q:200
        E:PL     Q:100
        E:PK    Q:100

$OFFTEXT
$SYSINCLUDE mpsgeset M2_2

*       Benchmark replication:
PX.L =1.; PY.L =1.; PLS.L =1.; PKS.L =1.;
TXL = 0.5;
TXK = 0.5;

M2_2.ITERLIM = 0;
$INCLUDE M2_2.GEN
SOLVE M2_2 USING MCP;
M2_2.ITERLIM = 2000;
```

Appendix C. Illustrative Model with Variable in Quantity, Productivity Spillovers

Listing C.1. MPSGE.jl Version of the Spillover Model.

```
using MPSGE,  Random, DataFrames, PlotlyJS
import JuMP.Containers:DenseAxisArray,@container
import Distributions.Uniform

function random_parameter(set; lower = 0, upper = 1)
    return @container([set], rand(Uniform(lower, upper)))
end

function create_data(N::Int; random_seed::Union{Int,Missing} = 1234)
    if !ismissing(random_seed)
        Random.seed!(random_seed)
    end
    regions = Symbol.("r",1:N)

    land_rent = random_parameter(regions) #n0
    capital_rent = random_parameter(regions) #k0
    wage = random_parameter(regions) #l0

    return (regions = regions, land_rent = land_rent, capital_rent =
     capital_rent, wage = wage)
end

regions, land_rent, capital_rent, wage = create_data(10)
initial_mu = 4
outputY = land_rent .+ capital_rent .+ wage #y0
# Assume all regions have balanced budgets at the benchmark
consumption = land_rent .+ capital_rent .+ wage
consumption_share = consumption./sum(consumption)

M1 = MPSGEModel()

@sectors(M1, begin
    Y[regions], (description = "Production in region")
    # Total consumption activity/Index, U x N
    C[regions], (description = "Consumption in region")
end)

@commodities(M1, begin
    P[regions], (description = "Regional supply")
    PN[regions], (description = "Rental index for land")
    PK, (description = "Captial price")
    PL[regions], (description = "Regional wage")
    PC[regions], (description = "Consumption price")
end)

@consumer(M1, RA[regions], description = "Representative agent for households
living in region")

@parameters(M1, begin
    productivity_index[regions], 1, (description = "Productivity index")
    μ, initial_μ, (description = "Logit parameter, elasticity for sorting")
    good_elasticity, 2, (description = "Good elasticity")
    γ, 1, (description = "Congestion spillover elasticity")
end)

@auxiliaries(M1, begin
    U[regions], (description = "Per-capita utility index (benchmark = 1)")
```

```
    N[regions], (description = "Population index (benchmark = 1)")
    congestion_externality[regions], (description = "Congestion spillover")
    price_index, (description = "Price index to fix as numeraire")
end)

set_start_value.(U, 1)
set_start_value.(N, 1)
set_start_value.(congestion_externality, 0)
set_start_value(price_index, 1)

for r∈regions
    @production(M1, Y[r], [s=0.5, kl=>s=1, t=0], begin
        @output(P[r], productivity_index[r] * outputY[r], t)
        @input(PN[r], land_rent[r], s)
        @input(PK, capital_rent[r], kl)
        @input(PL[r], wage[r], kl)
    end)
end

for r∈regions
    @production(M1, C[r], [s=good_elasticity, t=0], begin
        @output(PC[r], (1-congestion_externality[r])*consumption[r], t)
        [@input(P[rr], consumption_share[r]*outputY[rr], s) for rr∈regions]...
    end)
end

for r∈regions
    @demand(M1, RA[r], begin
        @final_demand(PC[r], outputY[r])
        @endowment(PL[r], wage[r]*N[r])
        @endowment(PK,    capital_rent[r]*N[r])
        @endowment(PN[r], land_rent[r])
    end)
end

# Per capital utility (economic well being)
for r∈regions
    @aux_constraint(M1, U[r],
        U[r] - RA[r]/(PC[r]*N[r]*consumption[r]) )
end

# Logit model determines the number of individuals located in region r:
for r∈regions
    @aux_constraint(M1, N[r],
        N[r] - exp(μ*U[r])/sum(consumption_share[rr]*exp(μ*U[rr]) for rr in
regions))
End
# A negative function of population, with a γ
for r∈regions
    @aux_constraint(M1, congestion_externality[r],
        congestion_externality[r] - (N[r]^γ - 1))
end
@aux_constraint(M1, price_index, price_index - sum(P[r]*outputY[r] for r in
regions) / sum(outputY[r] for r in regions))

# set_silent(M1)
solve!(M1, cumulative_iteration_limit=0)
bnchM1 = generate_report(M1)
bnchM1[!,:var] = string.(bnchM1[:,:var])
for r in regions
    push!(bnchM1, ["Prod_Ind$r" value(productivity_index[r])  0])
    push!(bnchM1, ["ConsShare$r" consumption_share[r] * value(M1[:N][r]) 0])
```

```
    push!(bnchM1, ["ConsShare0$r" consumption_share[r] 0])
end
price_index_bnch = sum(value(M1[:P][r])*outputY[r] for r in regions) /
sum(outputY[r] for r in regions)
solve!(M1, cumulative_iteration_limit=1000000)

set_value!.(M1[:productivity_index], random_parameter(regions, lower = 0.5,
upper = 1.5))
for r in regions
    fix(congestion_externality[r],0)
end
# fix(RA[:r1],consumption[:r1]) Can set the numeraire, or by default it will be
the largest consumer (r10)
solve!(M1, cumulative_iteration_limit=100000)

Shock_noextM1=generate_report(M1)
Shock_noextM1[!,:var] = string.(Shock_noextM1[:,:var])
for r in regions
    push!(Shock_noextM1, ["Prod_Ind$r" value(productivity_index[r])  0])
    push!(Shock_noextM1, ["ConsShare$r" consumption_share[r] * value(M1[:N][r])
0])
    push!(Shock_noextM1, ["ConsShare0$r" consumption_share[r] 0])
end
price_index_noext = sum(value(M1[:P][r])*outputY[r] for r in regions) /
sum(outputY[r] for r in regions)

data_frames_noext = []
# set_silent(M)
for μ in 0:.1:5
    set_value!(M1[:μ], μ)
    solve!(M1)

    tmp =  DataFrame(
        region = regions,
        population = [value(M1[:N][r])*consumption_share[r] for r in regions],
        U = [value(M1[:U][r]) for r in regions],
        N = [value(M1[:N][r]) for r in regions],
        Y = [value(M1[:Y][r]) for r in regions],
        C = [value(M1[:C][r]) for r in regions],
        productivity_index = [value(M1[:productivity_index][r]) for r in
regions],
        P = [value(M1[:P][r])/price_index_noext for r in regions],
        PN = [value(M1[:PN][r])/price_index_noext for r in regions],
        PL = [value(M1[:PL][r])/price_index_noext for r in regions],
        RA = [value(M1[:RA][r])/price_index_noext for r in regions],
        μ = value(M1[:μ]),
        good_elasticity = value(M1[:good_elasticity]),
    )
    # report(M1, regions, land_rent, capital_rent, wage)
    tmp[!,:μ] .= μ

    push!(data_frames_noext,tmp)
end

df_noext = vcat(data_frames_noext...);

# Finally, we plot the results using PlotlyJS
p = PlotlyJS.plot(df_noext, x=:μ, y=:N, color=:region, kind = "lines",
category_orders=attr(region=sort(df_noext[end-10:end,:],:N,
rev=true)[:,:region]),Layout(yaxis_title_text = "N: Index of
population",xaxis_title_text = "μ: logit parameter"))

p = PlotlyJS.plot(df_noext, x=:μ, y=:Y, color=:region, mode = "lines")
```

```
for r in regions
    unfix(congestion_externality[r])
    set_lower_bound(congestion_externality[r], -10)
    print(lower_bound(congestion_externality[r]))
end

solve!(M1, cumulative_iteration_limit=100000)
price_index_shock = sum(value(M1[:P][r])*outputY[r] for r in regions) /
sum(outputY[r] for r in regions)
ShockM1 = generate_report(M1)
ShockM1[!,:var] = string.(ShockM1[:,:var])
for r in regions
    push!(ShockM1, ["Prod_Ind$r" value(productivity_index[r])  0])
    push!(ShockM1, ["ConsShare$r" consumption_share[r] * value(M1[:N][r]) 0])
    push!(ShockM1, ["ConsShare0$r" consumption_share[r] 0])
end

data_frames = []
# set_silent(M)
for γ in 0:.1:5
    set_value!(M1[:γ], γ)
    solve!(M1)

    tmp =  DataFrame(
        region = regions,
        population = [value(M1[:N][r])*consumption_share[r] for r in regions],
        U = [value(M1[:U][r]) for r in regions],
        N = [value(M1[:N][r]) for r in regions],
        Y = [value(M1[:Y][r]) for r in regions],
        congestion_externality = [value(M1[:congestion_externality][r]) for r
in regions],
        productivity_index = [value(M1[:productivity_index][r]) for r in
regions],
        P = [value(M1[:P][r])/price_index_shock for r in regions],
        PN = [value(M1[:PN][r])/price_index_shock for r in regions],
        PL = [value(M1[:PL][r])/price_index_shock for r in regions],
        RA = [value(M1[:RA][r])/price_index_shock for r in regions],
        γ = value(M1[:γ]),
        good_elasticity = value(M1[:good_elasticity]),
    )
    # report(M1, regions, land_rent, capital_rent, wage)
    # tmp[!,:γ] .= γ

    push!(data_frames,tmp)
end
df = vcat(data_frames...);

p = PlotlyJS.plot(df, x=:γ, y=:congestion_externality, color=:region, mode =
"lines")
p = PlotlyJS.plot(df, x=:γ, y=:population, color=:region, mode = "lines")
p = PlotlyJS.plot(df, x=:γ, y=:N, color=:region, mode = "lines",
category_orders=attr(region=sort(df_noext[end-10:end,:],:N,
rev=true)[:,:region]),
 Layout(yaxis_title_text = "N: Index of population change",xaxis_title_text =
"γ: Congestion elasticity parameter",legend=attr(x=0.9, y=.98)))
```

Listing C.2. GAMS Version of the Spillover Model.

```
$title      Illustrative Sorting Model with Logit Distribution of Location
Preferences

set  r      Regions /r1*r10/;

alias (r,rr);

parameter   y0(r)          Benchmark output
            n0(r)          Benchmark land rent
            k0(r)          Benchmark capital rent
            l0(r)          Benchmark wage
            c0(r)          Benchmark consumption (aggregate)
            phi(r)         Productivity index
            theta(r)       Benchmark share of consumption in region r
            mu             Logit elasticity for sorting /2/
            γ              Spillover elasticity for disutility of growth /0.5/;

n0(r) = uniform(0,1);
k0(r) = uniform(0,1);
l0(r) = uniform(0,1);
y0(r) = n0(r) + k0(r) + l0(r);

*    Assume that all regions have balanced budgets:

c0(r) = n0(r) + k0(r) + l0(r);
theta(r) = c0(r) / sum(rr,c0(rr));
phi(r) = 1;

$ontext
$model:SORTING

$sectors:
    Y(r)    !      Production in region r
    C(r)    !      Consumption in region r

$commodities:
    P(r)    !      Regional supply
    PN(r)   !      Rental index for land
    PK      !      Capital price
    PL(r)   !      Regional wage
    PC(r)   !      Consumption price

$consumers:
    RA(r)   !      Representative agent for households living in r

$auxiliary:
    U(r)    !      Per-capita utility index (benchmark=1)
    N(r)    !      Population index (benchmark=1)
    Z(r)    !      Primal congestion externality multiplier
    LAMDA(r) !     Dual congestion externality multiplier

$prod:Y(r)  s:0.5 kl:1
    o:P(r)  q:(phi(r)*y0(r))
    i:PN(r) q:n0(r)
    i:PK    q:k0(r) kl:
    i:PL(r) q:l0(r) kl:

*    Dual impact of the congestion externality enters
*    the model through LAMDA(r):

$prod:C(r)  s:4
```

```
     o:PC(r) q:c0(r)                a:RA(r)   n:LAMDA(r)
     i:P(rr) q:(theta(r)*y0(rr))

$demand:RA(r)
     d:PC(r)
     e:PL(r) q:l0(r) r:N(r)
     e:PK    q:k0(r) r:N(r)
     e:PN(r) q:n0(r)

*    Primal impact of the externality:

     e:PC(r) q:(-1)  r:Z(r)

*    Add-hoc representation of an externality in MPSGEv1:
$constraint:Z(r)
     Z(r) =e= C(r) * c0(r) * LAMDA(r);

*    Congestion spillover due to population:

$constraint:LAMDA(r)
     LAMDA(r) =e= N(r)**γ - 1;

*    Per-capital utility (economic well-being):

$constraint:U(r)
     U(r)*N(r)*c0(r) =e= RA(r)/PC(r);

*    Logit model determimes the number of individuals
*    located in region r:

$constraint:N(r)
     N(r) =e= exp(mu*U(r))/sum(rr,theta(rr)*exp(mu*U(rr)));

$offtext
$sysinclude mpsgeset sorting

U.L(r) = 1;
N.L(r) = 1;

*    Verify that we are calibrated:

sorting.iterlim = 0;
$include sorting.gen
solve sorting using mcp;
abort$round(sorting.objval,3) "Benchmark replication error";

parameter   results(*,*,*) Model results,
            pnum           Numeraire price index;

$onechov >%gams.scrdir%report.gms
pnum = sum(r,P.L(r)*y0(r))/sum(r,y0(r));
results(%1,r,"theta") = theta(r)*N.L(r);
results(%1,r,"U") = U.L(r);
results(%1,r,"N") = N.L(r);
results(%1,r,"Y") = Y.L(r);
results(%1,r,"phi") = phi(r);
results(%1,r,"P") = P.L(r)/pnum;
results(%1,r,"PN") = PN.L(r)/pnum;
results(%1,r,"PL") = PL.L(r)/pnum;
$offecho

$batinclude %gams.scrdir%report "'bau'"
```

```
*    Draw a productivity shock ranging from -50% to +50%:

phi(r) = uniform(0.5,1.5);
sorting.iterlim = 10000;

LAMDA.UP(r) =  inf;
LAMDA.LO(r) = -inf;
Z.UP(r) =  inf;
Z.LO(r) = -inf;

set  gammaval /0*50/;

loop(gammaval,
     gamma = gammaval.val/10;

$include sorting.gen
     solve sorting using mcp;
     abort$round(sorting.objval,3) "Counterfactual solution error";

$batinclude %gams.scrdir%report gammaval
);
option decimals=3;
display results ;

execute_unload 'spill.gdx',results;
execute 'gdxxrw i=spill.gdx o=spill.xlsx par=results rng=PivotData!a2 cdim=0
intastext=n';
```

Appendix D. Large Dataset Example: U.S. Tariff Effect on Households in each State

Listing D.1. U.S. States Model with 5 Household Levels per State[58], Import Tariff Example[59].

```
HH = MPSGEModel()

@parameters(HH, begin
    ta[R,G], 0, (description = "Consumption Tax")
    ty[R,S], 0, (description = "Production Tax")
    tm[R,G], 0, (description = "Import Tax")
    tk[R,S], 0, (description = "Capital Tax")
    tfica[R,H], 0, (description = "FICA Labor Shares")
    tl[R,H], 0, (description = "Marginal Labor Tax")
end)

for r∈R, g∈G
    set_value!(ta[r,g], ta0[r,g])
    set_value!(ty[r,g], ty0[r,g])
    set_value!(tm[r,g], tm0[r,g])
    set_value!(tk[r,g], tk0[r])
end

for r∈R,h∈H
    set_value!(tfica[r,h], tfica0[r,h])
    set_value!(tl[r,h], tl0[r,h])
end

@sectors(HH, begin
    Y[R,S], (description = "Production")
    X[R,G], (description = "Disposition")
    A[R,G], (description = "Absorption")
    LS[R,H], (description = "Labor Supply")
    KS, (description = "Aggregate Capital Stock")
    C[R,H], (description = "Household Consumption")
    MS[R,M], (description = "Margin Supply")
    INVEST_DEMAND
    GOVT_DEMAND
end)

@commodities(HH, begin
    PA[R,G], (description = "Regional Market (input)")
    PY[R,G], (description = "Regional Market (output)")
    PD[R,G], (description = "Local Market Price")
    RK[R,S], (description = "Sectoral Rental Rate")
    RKS, (description = "Capital Stock")
    PM[R,M], (description = "Margin Price")
    PC[R,H], (description = "Consumer Price Index")
    PN[G], (description = "National Market Price for goods")
    PLS[R,H], (description = "Leisure Price")
    PL[R], (description = "Regional Wage Rate")
    PK, (description = "Aggregate return to capital")
    PFX, (description = "Foreign Exchange Rate")
```

58 Note: For a model this size, the use of PATH requires a (free) license installed, available with instructions via the https://pages.cs.wisc.edu/~ferris/path/

59 https://github.com/julia-mpsge/windc_household_model.jl/blob/main/src/model.jl has the full script of this model as a function.

```
        INVEST_COMMODITY
        GOVT_COMMODITY
    end)

    @consumers(HH, begin
        RA[R,H], (description = "Representative Agent")
        NYSE, (description = "Aggregate Capital Owner")
        INVEST, (description = "Aggregate Investor")
        GOVT, (description = "Aggregate Government")
        #ROW # if fint0 !=0
    end)

    @auxiliaries(HH, begin
        SAVRATE, (description = "Domestic Savings Rate")
        TRANS, (description = "Budget balance rationing variable")
        SSK, (description = "Steady-state capital stock")
        CPI, (description = "Consumer Price Index")
    end)

    set_start_value(SAVRATE, 1)
    set_start_value(TRANS, 1)
    set_start_value(SSK, 1)
    set_start_value(CPI, 1)

    for r∈R,ŝ∈S
        @production(HH, Y[r,ŝ], [t=0, s=0, va=>s=1], begin
            [@output(PY[r,g], ys0[r,ŝ,g], t, taxes = [Tax(GOVT, ty[r,ŝ])],
reference_price = 1-ty0[r,ŝ]) for g∈G]...
            [@input(PA[r,g], id0[r,g,ŝ], s) for g∈G]...
            @input(PL[r], ld0[r,ŝ], va)
            @input(RK[r,ŝ], kd0[r,ŝ], va, taxes=[Tax(GOVT, tk[r,ŝ])],
reference_price = 1+tk0[r])
        end)
    end

    for r∈R,g∈G
        @production(HH, X[r,g], [t=4, s=1], begin
            @output(PFX, x0[r,g] - rx0[r,g], t)
            @output(PN[g], xn0[r,g], t)
            @output(PD[r,g], xd0[r,g], t)
            @input(PY[r,g], s0[r,g], s)
        end)
    end

    for r∈R, g∈G
        @production(HH, A[r,g], [t=0, s=0, dm=>s=4, d=>dm=2], begin
            @output(PA[r,g], a0[r,g], t, taxes = [Tax(GOVT, ta[r,g])],
reference_price = 1-ta0[r,g])
            @output(PFX, rx0[r,g], t)
            @input(PN[g], nd0[r,g], d)
            @input(PD[r,g], dd0[r,g], d)
            @input(PFX, m0[r,g], dm, taxes = [Tax(GOVT, tm[r,g])],
reference_price = 1+tm0[r,g])
            [@input(PM[r,m], md0[r,m,g], s) for m∈M]...
        end)
    end

    for r∈R, m∈M
        @production(HH, MS[r,m], [t=0, s=0], begin
            @output(PM[r,m], sum(md0[r,m,gm] for gm∈GM), t)
            [@input(PN[gm], nm0[r,gm,m], s) for gm∈GM]...
            [@input(PD[r,gm], dm0[r,gm,m], s) for gm∈GM]...
```

```
        end)
    end

    for r∈R,h∈H
        @production(HH, C[r,h], [t=0, s=1], begin
            @output(PC[r,h], c0_h[r,h], t)
            [@input(PA[r,g], cd0_h[r,g,h], s) for g∈G]...
        end)
    end

    for r∈R, h∈H
        @production(HH, LS[r,h], [t=0, s=1], begin
            [@output(PL[q], le0[r,q,h], t, taxes=[Tax(GOVT, tl[r,h] +
tfica[r,h])], reference_price = 1-tl0[r,h]-tfica0[r,h]) for q∈Q]...
            @input(PLS[r,h], ls0[r,h], s)
        end)
    end

    @production(HH, KS, [t=etaK, s=1], begin
        [@output(RK[r,s], kd0[r,s], t) for r∈R, s∈S]...
        @input(RKS, sum(kd0[r,s] for r∈R, s∈S), s)
    end)

    for r∈R,h∈H
        @demand(HH, RA[r,h], begin
            @final_demand(PC[r,h], c0_h[r,h])
            @final_demand(PLS[r,h], lsr0[r,h])
            @endowment(PLS[r,h], ls0[r,h]+lsr0[r,h])
            @endowment(PFX, TRANS*sum(hhtrn0[r,h,trn] for trn∈TRN))
            @endowment(PLS[r,h], (tl[r,h] - tl_avg0[r,h])*sum(le0[r,q,h] for
q∈Q))
            @endowment(PK, ke0[r,h])
            @endowment(PFX, -sav0[r,h]*SAVRATE)
        end, elasticity = esubL[r,h])
    end

    @demand(HH, NYSE, begin
        @final_demand(PK, sum(yh0[r,g] for r∈R, g∈G)+ sum(kd0[r,s] for r∈R,
s∈S)*SSK)#)
        [@endowment(PY[r,g], yh0[r,g]) for r∈R, g∈G]...
        @endowment(RKS, SSK*sum(kd0[r,s] for r∈R, s∈S))
    end)

    @production(HH, INVEST_DEMAND, [t=0, s=0], begin
        @output(INVEST_COMMODITY, sum(i0[r,g] for r∈R, g∈G), t)
        [@input(PA[r,g], i0[r,g], s) for r∈R, g∈G]...
    end)

    @demand(HH, INVEST, begin
        #[@final_demand(PA[r,g], i0[r,g]) for r∈R, g∈G]...
        @final_demand(INVEST_COMMODITY, sum(i0[r,g] for r∈R, g∈G))
        @endowment(PFX, totsav0*SAVRATE)
        @endowment(PFX, fsav0)
    end)#, elasticity = 0)


    @production(HH, GOVT_DEMAND, [t=0, s= 1], begin
        @output(GOVT_COMMODITY, sum(g0[r,g] for r∈R,g∈G), t)
        [@input(PA[r,g], g0[r,g], s) for r∈R, g∈G]...

    end)
```

```
    @demand(HH, GOVT, begin
        #[@final_demand(PA[r,g], g0[r,g]) for r∈R, g∈G]...
        @final_demand(GOVT_COMMODITY, sum(g0[r,g] for r∈R,g∈G))
        @endowment(PFX, -TRANS*sum(trn0[r,h] for r∈R, h∈H))
        @endowment(PFX, govdef0)
        [@endowment(PLS[r,h], -(tl[r,h] - tl_avg0[r,h])*sum(le0[r,q,h] for
q∈Q)) for r∈R, h∈H]...
    end)

    #@demand(HH, ROW, begin
    #    @final_demand(PFX, fint0)
    #    @endowment(PK, fint0)
    #end)

    @aux_constraint(HH, SSK,
        sum(i0[r,g]*PA[r,g] for r∈R, g∈G) - sum(i0[r,g] for r∈R, g∈G) *RKS    )

    @aux_constraint(HH, SAVRATE,
        INVEST - sum(PA[r,g]*i0[r,g] for r∈R, g∈G)*SSK    )

    @aux_constraint(HH, TRANS,
        GOVT - sum(PA[r,g]*g0[r,g] for r∈R, g∈G)    )

    @aux_constraint(HH, CPI,
        CPI - sum(PC[r,h]*c0_h[r,h] for r∈R, h∈H)/sum(c0_h[r,h] for r∈R, h∈H)
)

fix(HH[:PFX], 1)

solve!(HH, cumulative_iteration_limit=0)

set_value!.(HH[:tm], .2)
solve!(HH)
tarrif_2 = generate_report(HH)
```

Supplementary Material

We include a set of files to enable interested people to run the three example models from the paper, and replicate the results as shown. The included README files give basic guidance on how to set up and run the model Julia scripts. The materials are included as a zip file: Supplementary material for MPSGE in Julia_Bringing Open-Source and Efficient Computation to concise CGE Model formation.zip